AppMaker

YOUR ASSISTANT PROGRAMMER!

B•O•W•E•R•S
Development

Serial # of original 1.1 disk: AM 925409

AppMaker™

Your Assistant Programmer

Version 1.5

For Apple® Macintosh®

B•O•W•E•R•S
Development

Credits

Software: Spec Bowers, Martin Kadansky, Walter Hunt

Documentation: Elisabeth Bayle, Rich Parker, Spec Bowers

Package Design: Robert Tinney, Tonya Price

Bowers Development Corporation
P.O. Box 9
Lincoln Center, MA 01773
(508) 369-8175

Contents

Introducing AppMaker 1.5

AppMaker™ 1.5 offers a variety of new features to make programming the Macintosh easier than ever. This latest version of AppMaker adds support for System 7.0, for the new THINK Class Library, and for MacApp 3.0. AppMaker 1.5 also supports the latest versions of these languages: THINK C 5.0, THINK Pascal 4.0, MPW C 3.2, MPW Pascal 3.2, and A/UX ANSI C. In addition, it has several enhancements to AppMaker itself.

AppMaker is now System 7.0-friendly, rather than just compatible. AppMaker 1.5 supports Apple Events and Stationery, and has a Balloon Help editor.

AppMaker 1.5 fully supports Version 1.1 of the THINK Class Library (TCL). The resource editor has been rewritten to create 'pane' resources for the user interface objects defined in the new TCL, including scroll panes, tables, and radio group panes. Panes can now be nested. The AppMaker Class Library extends the TCL by specifying font, size, and style in pane resources.

Enhancements include support for font, size, and style; movable modal dialogs; alignment options; and display of item numbers. AppMaker 1.5 merges the previous MacApp and non-MacApp versions of AppMaker into a single application.

For a complete list of AppMaker's new features, see Appendix B, "What's New in AppMaker 1.5."

Overview of AppMaker

AppMaker is a tool for making applications. It helps you design and code the user interface—the menus, windows, dialogs, and alerts—for a Macintosh application. Instead of typing thousands of lines of code, you simply point and click to design the results you want and AppMaker generates excellent source code to implement those results. AppMaker handles the routine details so that you can concentrate on the interesting parts of your application.

AppMaker is also a tool for enhancing existing applications. You can edit an application to rearrange or add to the user interface. You can, for instance, create new dialogs and then generate source code for just those new dialogs.

There are three major components to AppMaker:

- ☐ A user interface editor
- ☐ A source code generator
- ☐ A library of commonly-used routines

The **user interface editor** is a draw-like program that lets you create and edit resources for menus, windows, dialogs, alerts, and the many kinds of items in them. You can create normal, hierarchical, and pop-up menus, as well as picture menus. You can create both modal and modeless dialogs. In windows and dialogs, AppMaker supports all of the standard items such as buttons, check boxes, and radio buttons. AppMaker also supports less common items such as tool palettes, scrollable lists, pop-up menus, picture buttons, and custom sliders.

The **source code generator** generates complete source code which is ready to compile, link, and run without any editing. You can choose C or Pascal, THINK or MPW. AppMaker also supports object-oriented programming with the THINK Class Library or with MacApp. For MPW, AppMaker generates a .make file to compile and link the application. You can choose to generate source code for the complete application or for selected modules. After creating a new application, or when enhancing an existing application, you'll want to merge the generated source code with your own code. You'll find this easy to do since AppMaker's generated code is familiar-looking.

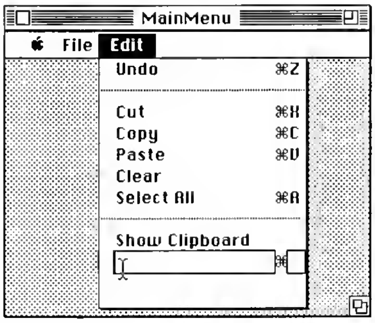
The **AppMaker library** helps implement much of the standard user interface, such as dragging windows, checking menu items, and handling radio buttons. There are also routines for dragging selection rectangles, handling pop-up menus, and for implementing tool palettes. AppMaker uses many of these routines to shorten and simplify the generated code. You'll also find some of the routines useful when writing your own code.

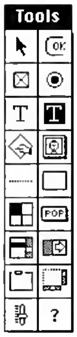
Making an Application

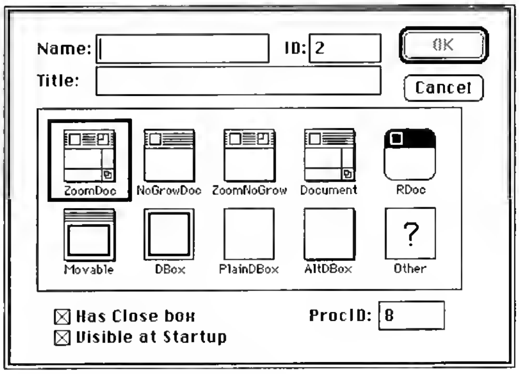
The following steps summarize the process of making an application with AppMaker:

1. Create a new application, or open an existing one to modify using AppMaker.

2. Using AppMaker tools, menus, and dialogs, point and click to create the user interface (menus, windows, dialogs, alerts, and items such as buttons, palettes, and lists).







Name:
ID: 2
Title:
☒ Has Close box
☒ Visible at Startup
ProcID: 8

3. Have AppMaker generate source code for your preferred language system.

Think C   Think Pascal   MPW C   MPW Pascal

4. Using your language system, compile and link.

   Your Application

These steps, *without any additional coding by you*, will create a working user interface. Normally you'll add code to make the application actually perform the desired functions called for by the menus, dialogs, etc.

If you're enhancing an existing application, you'll normally edit its user interface, perhaps adding new dialogs, generate source code for just the new user interface, then merge the generated code with your existing code and perhaps with new code, and compile and link. You'll be able to try out new ideas much faster than if you had to code them yourself.

About This Manual

This manual documents AppMaker, Your Assistant Programmer. It describes what AppMaker is and how to use it. The manual assumes that you are familiar with the Macintosh user interface. It also assumes that you are familiar with your language system (for example, MPW Pascal or THINK C) although that is not essential.

The manual does not assume that you are an experienced programmer. Even if you are not a programmer, you can use AppMaker to create a working application. Macintosh programming experience would be helpful for understanding how the generated application works and would be essential for adding your own code to the application.

This manual is divided into six chapters and four appendixes:

Chapter 1, "Getting Started," describes the hardware and software you need to run AppMaker, explains how to install AppMaker, and states the customer support policy.

Chapter 2, "Quick Tour," leads you step by step through AppMaker to create a sample application.

Chapter 3, "Creating and Editing the User Interface," describes the WYSIWYG (what you see is what you get) editor you use to create and edit an application's user interface. It describes how to work with menus, windows, dialogs, alerts, and all the items that they may contain. It also explains how to generate source code and how to compile and link.

Chapter 4, "An Example AppMaker Application," describes a sample application which was generated by AppMaker and to which we added code manually to create a working application.

Chapter 5, "Programming a Procedural Application," describes the structure of a procedural application generated by AppMaker, explains how we added code to complete the sample application, and documents the library routines which AppMaker supplies to simplify procedural code.

Chapter 6, "Programming a THINK Class Library Application," describes the structure of a TCL application generated by AppMaker, explains how we added code to complete the sample application, and documents the class library routines which AppMaker supplies to enhance the TCL.

Appendix A, "AppMaker's Menus," gives a short explanation of each of AppMaker's menu commands and keyboard commands.

Appendix B, "What's New in AppMaker 1.5," documents the new features of Version 1.5, and discusses compatibility with previous versions of AppMaker and with other programming tools.

Appendix C, "Standard Resources," documents the resources which AppMaker creates in any new application, and describes how you can customize AppMaker's stationery documents to change the standard resources.

Appendix D, "Customizing AppMaker's Code Generator," documents how you can customize AppMaker to generate code in your own particular style.

Chapter 1
Getting Started

This chapter provides basic information on how to start using AppMaker. It describes the hardware and software you need to run AppMaker.

It tells you how to install AppMaker onto your hard disk, and how to compile the AppMaker libraries so that you can use them later in your own projects.

The last section of this chapter describes our Customer Support policy and tells you how to contact us.

What You Need to Run AppMaker

You can run AppMaker on a very modest system:

- ☐ A Macintosh Plus or later Macintosh
- ☐ 800K disk drive
- ☐ System 6.0 or later

You can create applications and generate code on a very small system. However, if you want to compile and link the generated application, your language system (for example, MPW) will probably require more, such as 4Mb RAM and a hard disk.

AppMaker generates source code which is compatible with the following compilers:

- ☐ THINK C 5.0
- ☐ THINK Pascal 4.0
- ☐ MPW C 3.2
- ☐ MPW Pascal 3.2

AppMaker, its libraries, and the generated code are MultiFinder–aware, 32-bit clean, and System 7.0–friendly.

Installing AppMaker

AppMaker is distributed on two Compact Pro™ self–extracting archive disks. Each archive, when launched, expands to several folders and many files on your hard disk.

AppMaker.sea

1. Run AppMaker.sea from the Program Disk. Select a destination folder on your hard disk.

   The archive program will create a folder named AppMaker 1.5 containing several files and folders.

Libraries.sea

2. Run Libraries.sea from the Libraries Disk. Select the AppMaker 1.5 folder on your hard disk.

   The archive program will create several folders in the AppMaker 1.5 folder. When it has finished the contents of the folder will be:

   - ☐ The AppMaker 1.5 application.
   - ☐ A ResEdit TMPLs file—contains many TMPL resources that describe AppMaker's custom resource types.
   - ☐ Stationery folder—contains stationery documents for AppMaker's four language environments.
   - ☐ Examples folder—contains the Demo application for the Quick Tour in Chapter 2, and the AMReminder sample application for Chapters 4-6.
   - ☐ Libraries folder—contains library and class library routines for all of the languages supported by AppMaker.
   - ☐ CDEFs/MDEF folder—contains source code for the custom CDEFs and MDEF included with AppMaker.

Installing the AppMaker Library

AppMaker includes a library of routines which are needed by the AppMaker-generated code, and which are also useful to you when writing your own code. These libraries are provided in separate folders for each of the language systems supported by AppMaker. Installation instructions vary for each of the language systems, as described in the following sections.

To install the AppMaker library for THINK C:

1. Open the THINK folder in the “AppMaker 1.5:Libraries” folder.
2. Copy (or Move) the AMLibraryC folder and/or the AMClassLibC folder to whatever folder on your hard disk contains THINK C. AMLibraryC is used for procedural programming; AMClassLibC is used for TCL programming.

Note: For THINK C to work correctly with AppMaker-generated code, the AppMaker library folders MUST be in the same folder as the THINK C application itself.

To compile the AppMaker library for THINK C:

Before you can use the AMLibraryC.π project in your own projects, you must compile the AppMaker library modules.

1. Open your copy of the AMLibraryC folder. Double-click the AMLibraryC.π project to launch the THINK C compiler.
2. Choose Bring Up To Date from the Project menu to compile all of the modules, then choose Quit from the File menu to return to the Finder.

To compile the AppMaker class library for THINK C:

The AMClassLibC folder contains a project called Starter.π. As its name implies, it is a starting point for your own projects. It has the TCL files and the AppMaker class library files already added to it. You can save time later (at the expense of disk space) by compiling the Starter project now, and then later cloning the already–compiled project for your own projects.

1. Open your copy of the AMClassLibC folder. Move the Starter.π project to a folder on your hard disk which is outside your THINK C folder.
2. Double-click Starter.π to launch the THINK C compiler.
3. Choose Bring Up To Date from the Project menu to compile all of the modules, then choose Quit from the File menu to return to the Finder.
4. Move the Starter.π project back to the AMClassLibC folder or to any convenient folder on your hard disk. Later, when you start a new project you will make a copy of the Starter.π project file.

To install the AppMaker library for THINK Pascal:

1. Open the THINK folder in the "AppMaker 1.5:Libraries" folder.
2. Copy (or Move) the AMLibraryP folder and/or the AMClassLibP folder to whatever folder on your hard disk contains THINK Pascal. AMLibraryP is used for procedural programming; AMClassLibP is used for TCL programming.

To compile the AppMaker class library for THINK Pascal:

The AMClassLibP folder contains a project called Starter.π. As its name implies, it is a starting point for your own projects. It has the TCL files, and the AppMaker class library files already added to it. You can save time later (at the expense of disk space) by compiling the Starter project now, and then later cloning the already–compiled project for your own projects.

1. Open your copy of the AMClassLibP folder.
2. Double-click Starter.π to launch the THINK Pascal compiler.
3. Choose Build from the Run menu to compile all of the modules, then choose Quit from the File menu to return to the finder.

To install the AppMaker library for MPW C or MPW Pascal:

1. Open the MPW folder in the "AppMaker 1.5:Libraries" folder.
2. Copy (or Move) the AMLibraryC and/or the AMLibraryP folder to the Libraries folder in your MPW folder.
3. Copy (or Move) the UserStartup•AppMaker file to your MPW folder. The MPW commands in this file tell the C and Pascal compilers where to find the AppMaker library files.

To compile the AppMaker library for MPW C or MPW Pascal:

Before you can use the AppMaker library in your own application, you must compile its modules and build a library file.

1. Launch MPW.
2. Set the working directory to be the AMLibraryC (or AMLibraryP) folder.
3. Choose Build from MPW's Build menu. Type `AMLib.o`, then press the Return or Enter key.

   MPW will compile all of the library source files, then will run the Lib tool to combine the object files into a library file.
4. Quit the MPW application.

To install the AppMaker library for MacApp:

AppMaker 1.5 supports MacApp 2.0 with MPW C++, MPW Pascal, THINK Pascal, and p1 Modula-2. It currently supports MacApp 3.0 only in MPW C++. At this writing, the latest release of MacApp is 3.0B3. After MacApp becomes 3.0 final, we expect to release an AppMaker update that will also support MacApp 3.0 in Pascal.

The installation procedure for all of these MacApp environments is similar:

1. Open the MacApp 2.0 or MacApp 3.0 folder in the MPW or THINK folders in the Libraries folder.
2. Copy (or Move) the UAMLibraryM modules to your MacApp interfaces folder:
   - ☐ For C++, copy the .cp and .h files into the CIncludes (CPIncludes for MacApp 3.0) folder in your MacApp:Interfaces folder.
   - ☐ For Pascal, copy the .p files into the PInterfaces folder in your MacApp:Interfaces folder.
   - ☐ For Modula-2, copy both the .p files and the .DEF file into the PInterfaces folder in your MacApp:Interfaces folder.

Customer Support

At Bowers Development, our corporate policy is to treat customers the way we would like to be treated ourselves.

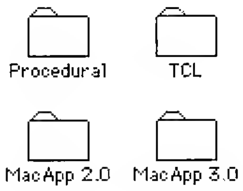
We are committed to offering powerful software and clear documentation, and to constantly improving our products. We listen to our customers. If you have any comments, suggestions, or questions please contact us through any of the following channels:

- ☐ AppleLink D1721, Bowers Development Corporation
- ☐ CompuServe 70731,3710
- ☐ America Online: Industry Connection forum keyword: BOWERS
- ☐ FAX (508) 369-8224
- ☐ Telephone (508) 369-8175 between 10 a.m. and 5 p.m. Eastern Time.

Chapter 2 Quick Tour

In this chapter you'll make a simple but non-trivial application, called Demo. It will have pull-down, hierarchical, and pop-up menus, a picture menu, a main window with a tool palette, a movable modal dialog, and a modeless dialog with several custom controls. In the process, you'll learn how to use most of AppMaker's features.

Creating the Demo Application



In the AppMaker 1.5:Examples folder, there is a Demo folder which contains a separate folder for each of the language environments that AppMaker supports: Procedural, THINK Class Library (TCL), MacApp 2.0, and MacApp 3.0.

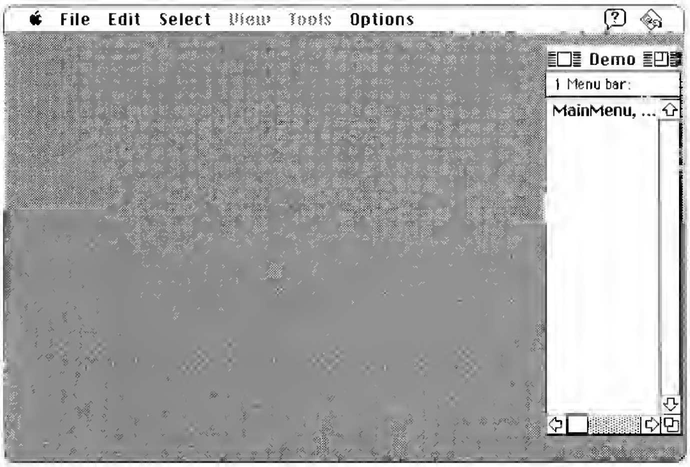
In each of these Demo folders, there is an AppMaker document named Demo or Demo.π.rsrc, which contains pictures, icons, and some other resources you'll use to create the Demo application. Some of these folders also contain THINK project files which will help you compile and link the Demo application.

Demo.π.rsrc is named in accordance with THINK naming conventions. If you are going to use MPW you might rename this document to just Demo. For brevity's sake, we will refer to this document as the Demo document in the remainder of this chapter.

1. Open the Demo folder for the language environment you wish to use.

2. Double-click the Demo document to launch AppMaker and open the Demo document.

   AppMaker displays the AppMaker screen:



Demo already contains several standard resources. Whenever AppMaker creates a new document it creates a main menu bar, a main window, and several standard alerts. (See Appendix B, Standard Resources for a list of the resources AppMaker creates in a new document.)

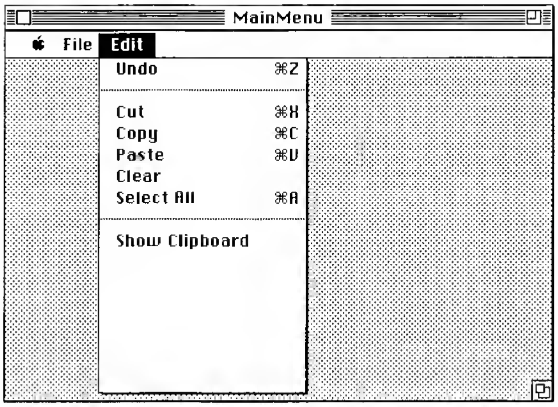
Looking at the Resources

1. In the Demo window, click MainMenu in the list, then click in the MainMenu window to activate it. Or, double-click MainMenu in the list.

   AppMaker displays the MainMenu window activated. The window contains a representation of a menu bar and behaves very much like the real menu bar.

2. Click the Edit title.

   A menu drops down:



   This Edit menu is the standard menu as described in *Inside Macintosh*, Chapter 2, "User Interface."

3. Examine the Apple ( ) and File menus by clicking their titles in turn. They, too, are standard menus.

Creating the Options Menu

You'll now add a new menu to the main menu bar. This new menu will have a hierarchical menu item and a divider line.

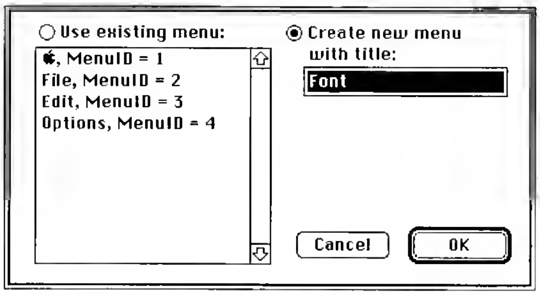
1. Press Command-K to create a new menu.

2. Type `Options`, then press Return to start typing the first menu item.

3. Type `Font`, then press the Tab key to begin typing in the command-key area.

4. Instead of typing a command-key, choose Create Submenu from the Edit menu, or press the Esc key.

   AppMaker displays a dialog box asking you which menu you want as the submenu:

   ○ Use existing menu:

   ```
   , MenuID = 1
   File, MenuID = 2
   Edit, MenuID = 3
   Options, MenuID = 4
   ```

   ● Create new menu with title:

   Font

   AppMaker has already provided `Font` as the title of the new submenu.

5. Click OK.

   AppMaker creates a SubMenus menu bar, and adds a Font menu to it.

   AppMaker displays the hierarchical indicator on the right edge of the menu item.

6. Press the Return key to start a new menu item. Type a hyphen (-), then press the Enter key to finish adding items.

   A menu item that begins with a hyphen becomes a divider line.

Creating the Things Menu

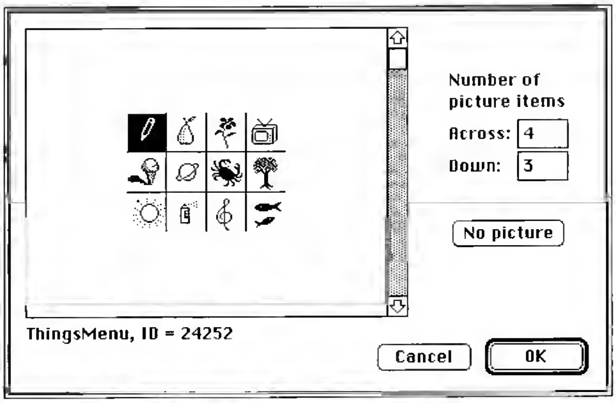
You'll now add a picture menu to the main menu bar. Creating a picture menu is just like creating a text menu, except that, after creating it, you assign a picture to go with it.

1. Press Command-K to create a new menu.

2. Type `Things`, then press the Enter key.

3. Choose Menu as Picture from the View menu.

   Since there is no picture currently associated with the menu, AppMaker displays a dialog box asking you to select a picture.



Number of picture items

Across: 4

Down: 3

ThingsMenu, ID = 24252

4. Type 4 in the Picture Items Across field.

5. Press the Tab key, then Type 3 in the Picture Items Down field.

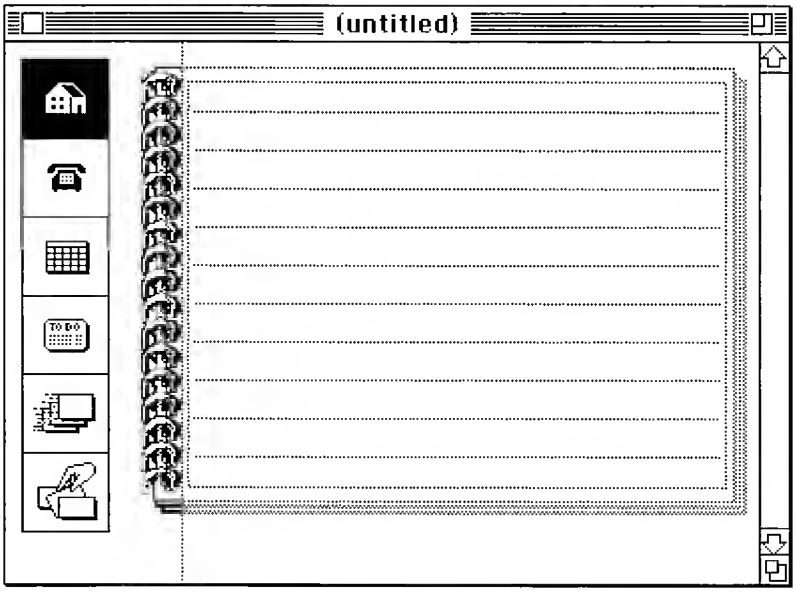
6. Click OK.

   AppMaker displays the menu, in picture form, in the MainMenu window.

Creating the Main Window

You'll now create the Demo application's main window.

Use the following picture as a guide for what you will create in the next several sections:



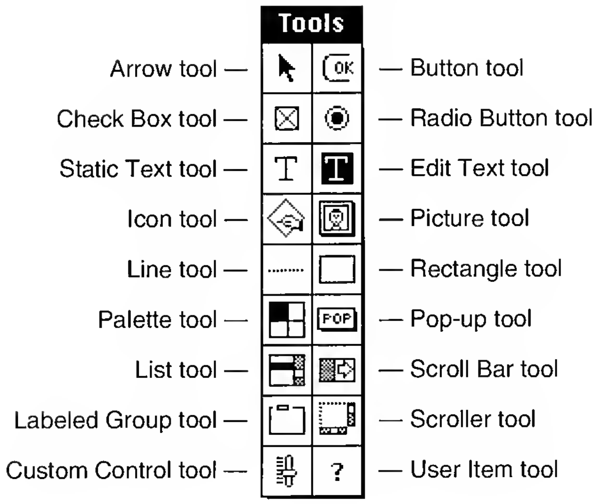
1. From the Select menu, choose Windows. Or, press Command-2, the keyboard shortcut.

   In the Demo window, AppMaker displays a list of currently defined windows for the application.

2. In the Demo window, double-click MainWindow to select it and activate it.

   The Tools menu is now enabled.

3. Pull down the Tools menu to see the various tools that you can use to create items in windows, dialogs, and alerts.



You'll use some of these tools to design the main window and some, later, to design dialogs.

Creating the Tool Palette

Many applications have a **tool palette,** a picture of various tools which the user can select by clicking. AppMaker supplies a custom CDEF (Control DEFinition) for easily implementing palettes.

1. Select the Palette tool from the Tools menu.

2. Position the crosshair near the upper-left corner of the main window, then click.

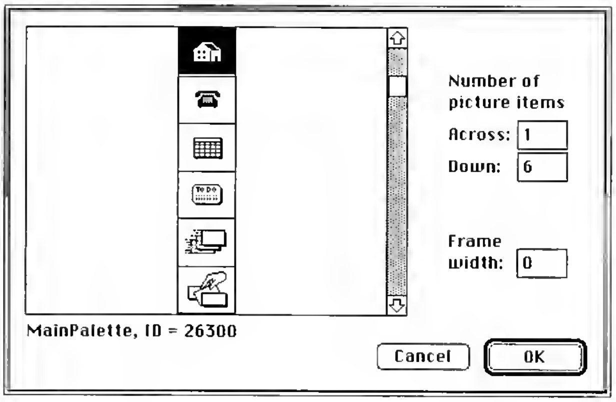
   AppMaker displays a dialog box that lets you select a picture and to set its "dimensions." The upper-left of the picture is highlighted to indicate the current dimensions.

3. Click in the scroll bar's down arrow to show the second picture.

4. Type 1 in the Picture Items Across field.

5. Press the Tab key, then type 6 in the Picture Items Down field.

Your screen should look like this:



Number of picture items

Across: 1

Down: 6

Frame width: 0

MainPalette, ID = 26300

6. Click OK to create a 1x6 palette in the main window.

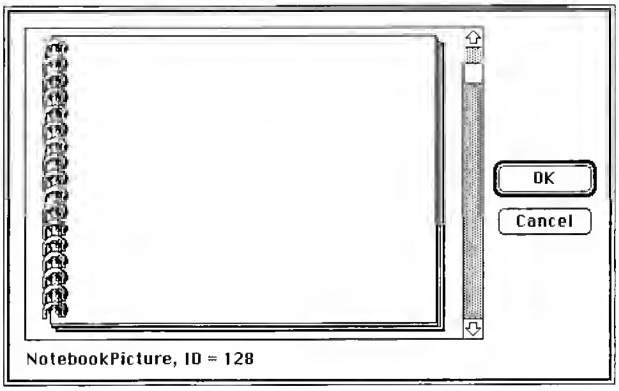
Creating the Background Picture

To make this window look a little more interesting, you'll add a background picture of a spiral notebook.

1. Select the Picture tool from the Tools menu.

2. Position the crosshair where you want the upper-left corner of the spiral notebook to appear, then click.

AppMaker displays a dialog box which lets you select a picture from among those already in Demo's resource file.



4. Click in the scroll bar to select the spiral notebook picture.

5. Click OK to add the spiral notebook picture to the main window.

6. If the picture is not exactly where you want it, drag the picture to reposition it. Or, press an arrow key to nudge the picture.

Creating the Scroll Pane (for TCL or MacApp)

A Scroll Pane (Scroller in MacApp terminology) marks a rectangular section of the window which can scroll vertically and/or horizontally. It can have both a vertical and a horizontal scroll bar or no scroll bar at all.

AppMaker supports scroll panes in TCL and MacApp, but not in procedural languages. Instead, create a scroll bar as described in the next section.

1. Select the Scroller tool.

   The pointer changes to a crosshair.

2. Position the crosshair near the left edge of the notebook picture, just to the right of the notebook's spiral, but near the top edge of the window.

3. Drag the pointer down to the bottom right corner of the window.

   When you release the mouse, AppMaker will draw a gray frame around the scroll pane, and draw a vertical scroll bar along its right edge. (Part of the background picture will also be redrawn in gray to indicate that it is "underneath" the scroll pane.)

   Later if you want, you can add a horizontal scroll bar.

If you later resize the window, AppMaker will automatically resize the scroll pane.

Creating the Scroll Bar (for Procedural Languages)

In place of a scroll pane which is supported by TCL and MacApp, in procedural languages AppMaker supports scroll bars along the edges of windows.

1. Select the Scroll Bar tool.

   The pointer changes to a crosshair.

2. Position the crosshair near the upper right corner of the main window, where you want the top of the scroll bar to appear, then click the mouse button.

   AppMaker creates a standard vertical scroll bar. Its left edge is 15 pixels in from the right edge of the window. Its bottom is just above where the window's grow icon will be. Its top is where you clicked. You control where the top goes; AppMaker takes care of the rest for you.

If you later resize the window, AppMaker will automatically move and resize the scroll bar.

Creating the Edit Text Field

1. Select the Edit Text tool from the Tools menu.

2. Position the pointer near the upper-left corner of the spiral notebook picture. Drag to near the lower-right corner of the picture.

   When you release the mouse button, you'll see a multi-line edit text field.

3. From the Edit menu, choose Text Style.

   AppMaker displays a dialog allowing you to select font, size, style, and justification.

4. Select New York, 14, Italic, and Centered, then click OK.

You've learned how to create a main window with a vertical and a horizontal scroll bar, a background picture, an edit text field, and a tool palette.

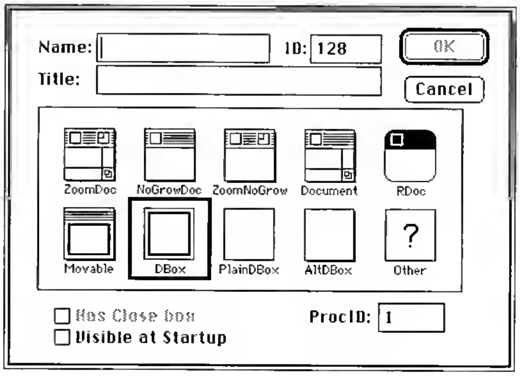
Creating the Communicate Dialog

1. From the Select menu, choose Dialogs. Or, press Command-3, the keyboard shortcut.

   In the Demo window, AppMaker displays a list of the currently defined dialogs.

2. From the Edit menu, choose Create Dialog. Or, press Command-K, the keyboard shortcut.

   AppMaker displays a dialog box which lets you select a window type, a name, and an ID for the new dialog.



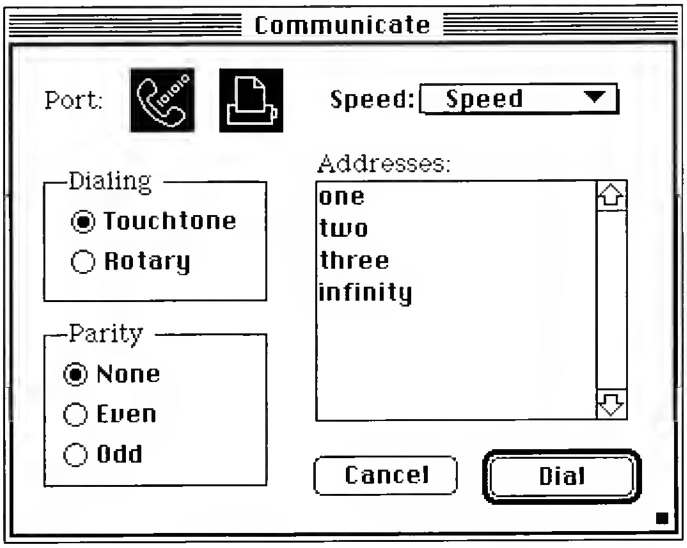
3. Type `Communicate` in the Name field.

4. Instead of accepting the default window type (DBox), click Movable in the palette of window types to make this a movable modal dialog.

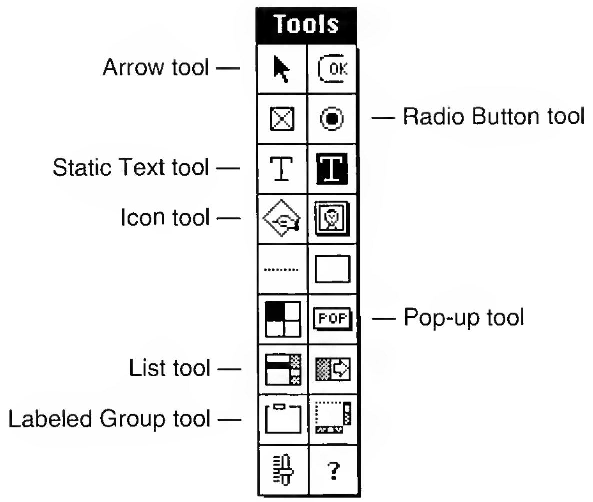
5. Click OK to create the new dialog.

   AppMaker displays the new dialog and selects it so that you can start adding items. The OK and Cancel buttons are predefined as items 1 and 2, respectively.

Use the following picture as a guide to help you position items in the Communicate dialog, as described in the following steps.



Pull down the Tools menu.



In the previous section, you used the Palette, Picture, Scroll Pane (or Scroll Bar), and Edit Text tools. In the next several sections, you'll use the Static Text, Labeled Group, Icon, Pop-up, Radio Button, List, and Arrow tools.

Creating the Icon Buttons

An icon is normally just decorative; clicking it does nothing. An icon button is an active element; it responds to mouse-down events.

AppMaker treats icon buttons in procedural dialogs in the same way as radio buttons: the selected icon is highlighted and the other icons in the group are displayed in their normal state. For TCL and MacApp, AppMaker treats icon buttons as push-buttons: they are highlighted while the mouse button is held down, but they return to normal when the mouse button is released.

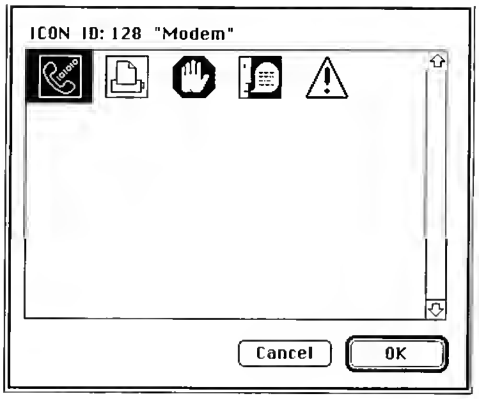
First, create a label for the icons:

1. Select the Static Text tool from the Tools menu.
2. Click near the upper left of the dialog window and type `Port:`

To create the icon buttons:

1. Select the Icon tool, then click where you want the upper-left corner of the Modem icon button.

   AppMaker displays a list of available icons.



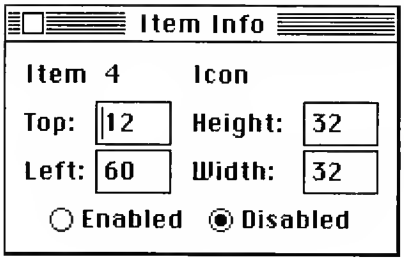
2. Click OK to create the "Modem" icon button.

When you create an icon it is disabled by default; at run-time it won't respond to mouse clicks. This is appropriate since most icons are for display only. To turn an inactive icon into an active icon button, you must enable it.

To enable this icon:

1. Choose Item Info from the View menu. Or, press Command-I, the keyboard shortcut.

   AppMaker displays the Item Info dialog:

   Item Info

   Item 4 Icon

   Top: 12 Height: 32

   Left: 60 Width: 32

   ○ Enabled [✓ Disabled]

   This dialog displays the item number (determined by the order in which the item was created), the kind of item, its top and left coordinates, its height and width, and whether it is enabled or disabled. For MacApp and TCL, the Item Info dialog displays additional information.

2. Click the Enabled radio button to enable the "Modem" icon.

   The icon is inverted to indicate that it is an icon button, not just a static icon.

There is a quicker way to create an icon button. If you hold down the Option key when you create an item, its enabled/disabled status is opposite to normal.

To create an enabled icon button the quick way:

1. Click in the Communicate window to select it. (The Item Info dialog is currently selected.)

2. Hold down the Option key and click where you want the "Printer" icon button.

3. Select the "Printer" icon, then click OK to create the second icon button.

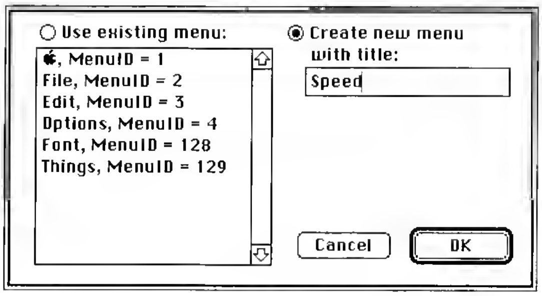
   AppMaker displays the icon inverted to indicate that it is an enabled icon button.

Creating the Speed Pop-up Menu

A pop-up menu has two parts: a label which identifies it and a box in which the menu "pops-up."

1. Select the Pop-up tool from the Tools menu.
2. Click where you want the upper-left corner of the Pop-up menu.

   AppMaker displays a dialog box asking you to select an existing menu or create a new menu:



○ Use existing menu:

, MenuID = 1
File, MenuID = 2
Edit, MenuID = 3
Options, MenuID = 4
Font, MenuID = 128
Things, MenuID = 129

[✓ Create new menu with title:] Speed

3. Type Speed, then click OK or press the Enter key.

   AppMaker displays the Speed label and the pop-up box.

Later, you'll add items to this pop-up menu.

Creating the Radio Button Groups

MacApp and the TCL provide special classes for handling radio buttons as a group. These classes implement the standard behavior of radio buttons and optionally provide a visual grouping around the buttons. AppMaker provides a similar tool for use with procedural languages.

1. Select the Labeled Group tool from the Tools menu. Drag a rectangle large enough to hold two radio buttons.

   The pointer changes to an I-beam so that you can type a label.

2. Type `Dialing`, then press Enter.

   AppMaker displays a frame with a label.

3. Select the Radio Button tool.

4. Click where you want the Touch-tone radio button.

5. Type `Touch-tone`.

6. Click where you want the Rotary radio button.

7. Type `Rotary`.

In a similar fashion, create a second set of radio buttons in a group labeled Parity, with three radio buttons labeled None, Even, and Odd.

AppMaker recognizes that these three radio buttons are in a separate group from the Touch-tone and Rotary buttons. The generated code will handle the groups appropriately.

Creating the Address List

For procedural languages, AppMaker uses the Macintosh List Manager to implement scrolling lists. For MacApp and TCL, AppMaker uses the more flexible and powerful classes that they provide for managing scrollable lists.

AppMaker creates a one-column list with a vertical scroll bar. This is the most common form of list, but you can create other kinds of lists by modifying the AppMaker library or the generated code.

First, create a label for the list:

1. Select the Static Text tool from the Tools menu.

2. Click where you want the Addresses label, then type `Addresses:`

To create a list (for procedural languages):

1. Select the List tool.
2. Position the pointer where you want the upper-left corner of the List. Drag to the lower-right corner.

   When you release the mouse button, you'll see a scrollable list.

To create a list (for TCL):

For TCL, a scrollable list comprises three parts: a frame (CBorder), a scroll pane (CScrollPane), and the list itself (CArrayPane or CTable).

1. Select the Rectangle tool.
2. Position the pointer where you want the upper-left corner of the list, then drag to the lower-right corner.
3. Select the Scroll Pane tool.
4. Position the pointer on the upper-left corner of the frame, then drag to the lower-right corner.

   AppMaker creates a scroll pane with a vertical scroll bar. The scroll bar is drawn inside the scroll pane, as the TCL will create it. You can later add a horizontal scroll bar if you want.
5. Press Command-G to turn off AppMaker's Grid so that you can precisely position the list within the scroll pane.
6. Select the List tool.
7. Position the pointer 1 pixel down and to the right of the upper-left corner of the scroll pane.
8. Drag to 1 pixel above the bottom of the scroll pane, and 1 pixel to the left of the vertical scroll bar.
9. Press Command-G to turn the Grid back on.

To create a list (for MacApp):

For MacApp, a scrollable list comprises three parts: a frame (TControl), a scroller (TScroller), and the list itself (TTextListView).

1. Select the Rectangle tool.
2. Position the pointer where you want the upper-left corner of the list, then drag to the lower-right corner.

3. Press Command-G to turn off AppMaker's grid so that you can precisely position the scroller within the frame.

4. Select the Scroller tool.

5. Position the pointer 1 pixel down and to the right of the upper-left corner of the frame.

6. Drag toward the lower-right corner of the frame. Leave room for the vertical scroll bar. MacApp creates its scroll bars outside the scroller.

   AppMaker creates a scroller with a vertical scroll bar. The scroll bar is drawn outside the scroller, as MacApp will create it. You can later add a horizontal scroll bar if you want.

7. Select the List tool.

8. Drag from the upper-left corner of the scroller to the lower-right corner of the scroller (but not its scroll bar).

9. Press Command-G to turn the Grid back on.

Renaming the OK Button

1. Select the Arrow tool.

2. Click the OK button to select it.

   The button is outlined to indicate it is selected and has a handle in the lower-right corner so that you can reshape it.

3. Click again to select the button's title for editing.

   The pointer changes to an I-beam.

4. Double-click to select the word OK.

   Note: Steps 3 and 4 can be combined into one double-click. Steps 2, 3, and 4 can be combined into one triple-click.

5. Type `Dial`, then press Enter to stop editing.

You've learned how to create a modal dialog and how to create static text labels, icon buttons, a pop-up menu, radio button sets, and a scrollable list. You've also learned how to edit an item's title.

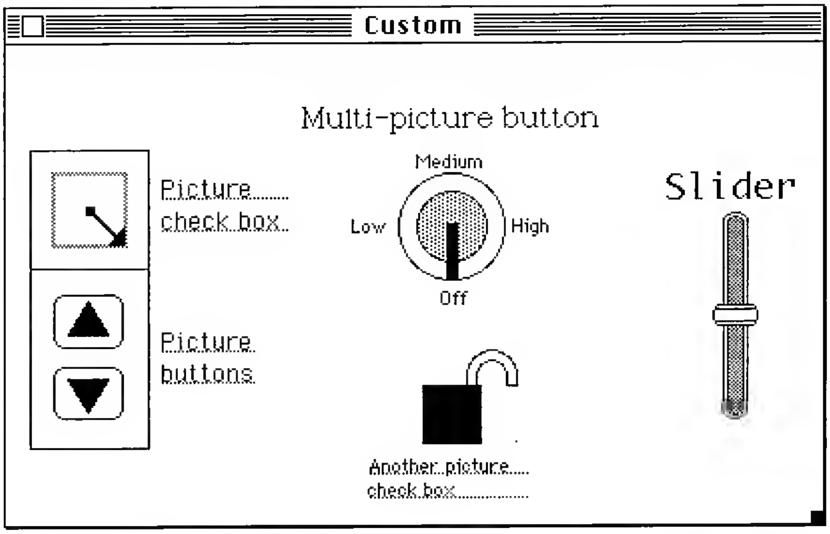
Creating the Custom Dialog

Custom controls give an application a professional touch. In this section, you will create two kinds of custom controls in a modeless dialog.

1. Click in the Demo window to select it.

2. Double-click Custom in the list to select it and to activate its window.

   AppMaker displays the Custom dialog and selects it so that you can start adding items. Use the following picture as a guide to help you add items to the dialog:



The Custom dialog already shows several examples of custom controls. You'll complete this dialog using the Custom Control tool. With this tool, you can use pictures in place of the standard Macintosh buttons, check boxes, and radio buttons. You can also create multi-picture buttons, and custom sliders.

Creating the Padlock Check Box

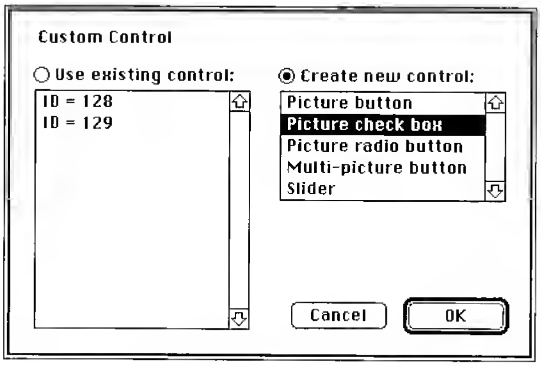
A picture check box behaves like a standard check box, except that it is composed of two pictures: one representing the checked or "on" state; the other representing the unchecked or "off" state. When you click on a check box or picture check box it changes between the two states.

1. Select the Custom Control tool from the Tools menu.

   The pointer changes to a crosshair when it is in the Controls window.

2. Click below the multi-picture button, where you want the upper-left corner of the padlock to appear.

   AppMaker displays a dialog box asking you to select an existing control or create a new custom control.


Custom Control

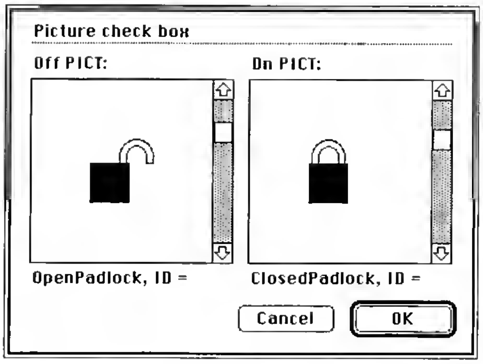
Use existing control: [✓ Create new control:]

ID = 128
ID = 129

Picture button
Picture check box
Picture radio button
Multi-picture button
Slider

3. Select Picture Check Box from the right-hand list, then click OK.

   AppMaker displays a dialog box asking for information about the picture check box:



4. Scroll the left-hand list until the "off" picture shown above is displayed.

5. Scroll the right-hand list until the "on" picture shown above is displayed.

6. Click OK.

   AppMaker displays the padlock picture check box, and automatically puts the appropriate CDEF into your application.

Creating the Slider

A slider is a *dial*, a control that a user can adjust to select from a range of values. The most common dial in Macintosh software is the standard scroll bar, but with AppMaker you can create a slider from any two pictures—one representing the background, and the other representing the indicator, or thumb.

1. Select the Custom Control tool from the Tools menu.

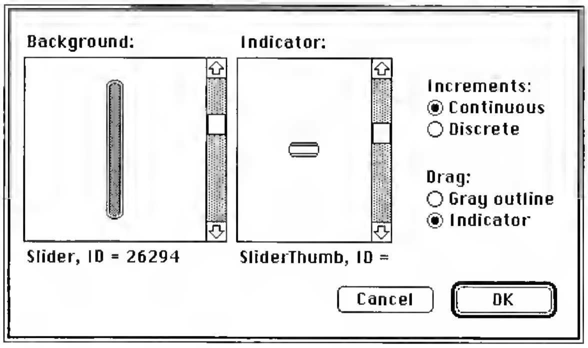
   The pointer changes to a crosshair when it is in the Controls window.

2. Click to the right of the multi-picture button, where you want the upper-left corner of the slider to appear.

   AppMaker displays a dialog box asking you to select an existing control or create a new custom control.

3. Select Slider from the right-hand list, then click OK.

   AppMaker displays a dialog box asking for information about the slider.



Background: Slider, ID = 26294

Indicator: SliderThumb, ID =

Increments: [✓ Continuous] Discrete

Drag: Gray outline [✓ Indicator]

4. Scroll the left-hand list until the background picture shown above is displayed.

5. Scroll the right-hand list until the indicator picture shown above is displayed.

6. Click Indicator to select a Drag option, then click OK.

   AppMaker displays the slider in the Controls window.

Creating the Brush Shape Dialog

Most dialogs have multiple items. They respond to events until a particular button is pressed to dismiss the dialog. A one-choice dialog has a single item (usually a palette), and clicking in that item dismisses the dialog.

1. Click in the Demo window to select it.
2. Choose Create Dialog from the Edit menu. Or, press Command-K, the keyboard shortcut.
3. Type `Brush Shape` in the Name field, then press the Enter key to create the dialog.
4. Using the Arrow tool, drag a selection rectangle around the OK and Cancel buttons. Press the Delete key to delete the two buttons.

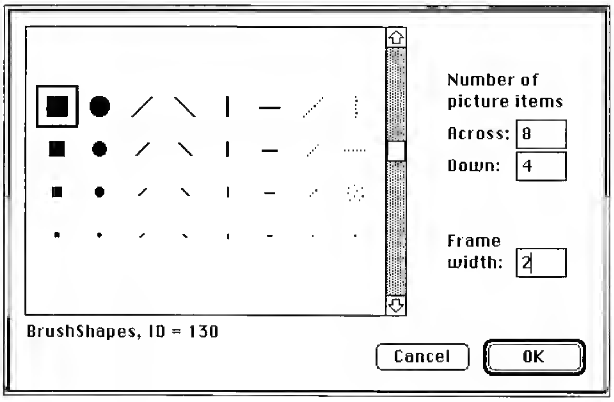
To create the brush shape palette:

1. Select the Palette tool from the Tools menu.
2. Click near the upper-left corner of the dialog window.

   AppMaker displays a dialog box for creating a MacPaint-style palette.
3. Click twice in the scroll bar's down arrow to show the Brush Shape picture.
4. Type `8` in the Items Across field.
5. Press the Tab key, then type `4` in the Down field.
6. Press the Tab key, then type `2` in the Frame Width field.

   The number 2 in the Frame Width field indicates the pen width of the frame around the selected item. A zero (0) value, as used for the palette in the main window, indicates that the selected item is inverted rather than framed.

Your screen should look like this:


Number of picture items
Across: 8
Down: 4
Frame width: 2
BrushShapes, ID = 130

7. Click OK to create the palette.

8. Position the pointer in the lower-right corner of the window's frame. Drag the window's corner upward and to the left for a snug fit around the palette.

Finishing the Menus

1. From the Select menu, choose Menus. Or, press Command-1, the keyboard shortcut.

   The Demo window now displays the list of menu bars.

2. Double-click MainMenu to open the MainMenu window and activate it.

3. Click in the Options title to drop down its menu.

4. Click below the divider line to start typing a new menu item.

5. Type `Communicate`, then press the Return key to start typing the next menu item.

   AppMaker displays an alert box asking you if you want an ellipsis to be added to the Communicate menu item. A menu item that invokes a dialog usually has an ellipsis after it.

6. Press the Enter key to add an ellipsis to the Communicate menu item and to start typing the next menu item.

7. Type `Custom…`, then press the Return key to start typing the next menu item.

   Note: You can type an ellipsis (…) either by pressing Option-semicolon (;) or by typing three periods. The appearance on the screen is the same and AppMaker recognizes either method as an ellipsis.

8. Type `Brush Shape…`, then press the Tab key to type a Command-key equivalent.

9. Press the `b` key to define a Command-key equivalent for the Brush Shape menu item, then press the Enter key to stop adding menu items.

   Note: AppMaker automatically converts the key you type into upper case. You don't have to hold down the Shift key.

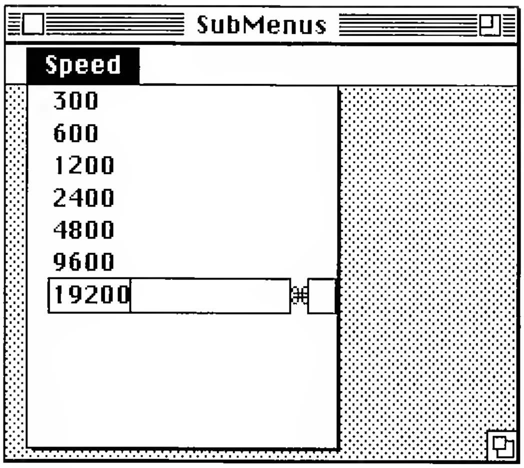
10. For TCL or MacApp users: Both TCL and MacApp require that most menu items have command numbers. To add command numbers, click and type in the command number section of each menu item. (If the command number section is not visible, then choose Show Command Numbers from AppMaker's Options menu.)

    MacApp reserves command numbers up to 999; the TCL reserves command numbers up to 1023. AppMaker defines some command numbers in the 1100-2000 range. Enter 2001 for the Font menu item, and 2002, 2003, 2004 for the three other enabled menu items. You don't need a command number for the disabled divider line.

Completing the Speed Pop-up Menu

When you created the Speed pop-up menu, AppMaker added it to the SubMenus menu bar. Now, you'll finish the menu by adding items to it.

1. Click in the Demo window to activate it.
2. Double-click SubMenus to open the SubMenus window and activate it.
3. Click in the Speed title to drop down its menu.
4. Fill in the Speed menu with speeds from 300 to 19200, pressing the Return key after typing each speed, as shown below:



Note: AppMaker displays a very tall menu so that you'll have room to add many menu items. At run-time, the menu will be only as tall as the items it contains.

5. Press the Enter key when you're finished typing all of the speeds.

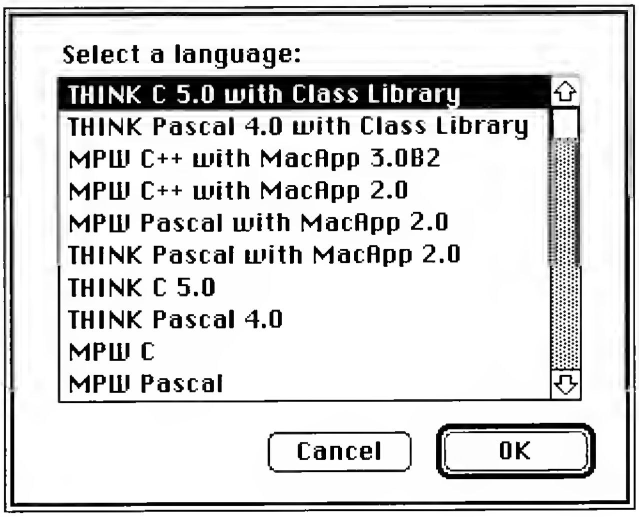
Generating Source Code for the Demo Application

1. Take a moment to examine your work. Look at each of the menus in the main menu and in the submenus. Look at the main window and at each of the dialogs.

   In a matter of minutes you have created the user interface for a non-trivial application. In just a few more minutes, you'll have an executable application.

2. From the File menu, choose Language.

   AppMaker displays a dialog box asking you to select a language system:



Select a language:

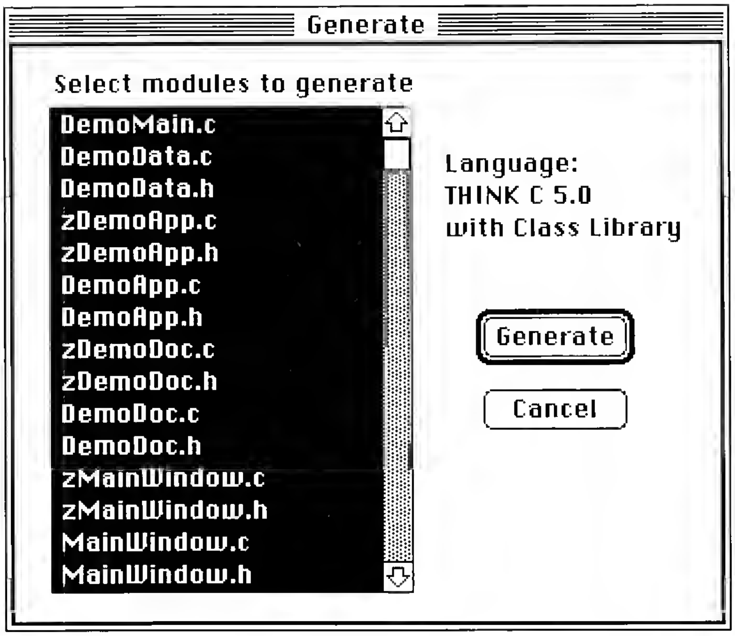
THINK C 5.0 with Class Library
THINK Pascal 4.0 with Class Library
MPW C++ with MacApp 3.0B2
MPW C++ with MacApp 2.0
MPW Pascal with MacApp 2.0
THINK Pascal with MacApp 2.0
THINK C 5.0
THINK Pascal 4.0
MPW C
MPW Pascal

3. Select your preferred language system, then click OK.

   Note: Each of the language categories (Procedural, TCL, MacApp 2.0, MacApp 3.0) requires different kinds of resources. Select a language from the category corresponding to the Demo folder you chose to start this Quick Tour.

4. From the File menu, choose Generate.

   AppMaker displays a dialog box something like this one:



This dialog varies slightly from one language system to another.

In this dialog, you can select one, many, or all of the files to generate. If you were creating an enhancement for an existing application, you might select just one of the files. Since this is a new application, you should select all files, which is the default selection.

5. Click Generate to generate source code.

   AppMaker deselects each file after it is generated until all of the selected files have been generated.

6. From the File menu, choose Quit.

   AppMaker displays a standard Save dialog box.

7. Click Save or press the Enter key to save your work.

Compiling and Linking the Demo Application

AppMaker has generated a complete set of source code files, ready to compile, link, and run. Now you'll use your language system to compile them and link them with the AppMaker library routines and with the standard system libraries. Following are separate instructions for each of the language systems.

For THINK C with Class Library:

The Demo folder contains a THINK C project called Demo.π. This project already references the standard TCL files and the AppMaker class library files and already uses the AppMaker-generated files. You'll just compile, link, and run.

1. In the Finder, double-click Demo.π to launch THINK C and open the project.
2. From the Project menu, choose Run.

   THINK C will compile all of the files, then link, and run the Demo application.

For THINK C (without Class Library):

The Demo folder contains a THINK C project called Demo.π. This project already references the standard THINK libraries and the AppMaker library routines, and already uses the AppMaker-generated files. You'll just compile, link, and run.

Note: If you did not compile the AppMaker Library when you installed it, do so now. For installation information, see the earlier section, "Installing the AppMaker Library."

1. In the Finder, double-click Demo.π to launch THINK C and open the project.
2. From the Project menu, choose Run.

   THINK C will compile all of the files, then link, and run the Demo application.

For THINK Pascal or THINK Pascal with Class Library:

The Demo folder contains a project called DemoP.π. This project already references the standard THINK libraries and the AppMaker library routines and already uses the AppMaker-generated files. You'll just compile, link, and run.

1. In the Finder, double-click DemoP.π to launch THINK Pascal and open the project.

2. From the Run menu, choose Go.

   THINK Pascal will compile all of the files, then link, and run the Demo application.

For MPW Pascal or MPW C:

1. In the Finder, double-click Demo.make to launch MPW and to select the Demo folder as the current directory.

   Note: You aren't going to edit this file (or any of the files). Double-clicking is just the most convenient way to launch MPW. You could have launched MPW by double-clicking any of the generated files or double-clicking MPW Shell itself.

2. Close the Demo.make window.

3. From the Build menu, choose Build.

4. Type `Demo`, then click OK.

   MPW will compile and link your Demo program, changing your AppMaker document into an application. When it is finished, the last line of the Worksheet will contain the single word `Demo` as a command to run the finished program.

5. Press the Enter key to run your Demo.

For MPW with MacApp:

1. In the Finder, double-click Demo.MAMake to launch MPW and to select the Demo folder as the current directory.

   Note: You aren't going to edit this file (or any of the files). Double-clicking is just the most convenient way to launch MPW. You could have launched MPW by double-clicking any of the generated files or double-clicking MPW Shell itself.

2. Close the Demo.MAMake window.

3. In MPW's Worksheet, type MABuild Demo then press the Enter key.

   MPW will compile and link your Demo program. When it is finished, it will show a line you can execute to run Demo.

Chapter 3
Creating and Editing the User Interface

To create a prototype of an application, you create its user interface and have AppMaker generate code for the user interface, as described in this chapter. The prototype will have all the aspects of the user interface (menus and windows, icons and static text, and so on), but no functions yet that make the application perform the tasks of a spreadsheet, database, word processor, or whatever it is designed to do. In AppMaker's generated code, there are placemarkers that tell you where to add the application-specific code to make the application function. The process for doing this is described in Chapter 5, "Programming a Procedural Application," and Chapter 6, "Programming a THINK Class Library Application." When this process is complete, you have a working application; in other words, you don't throw away the prototype, you just add the necessary code to make it functionally complete.

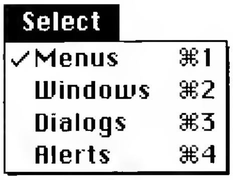
When needed, you can change or add to the user interface of your application, just by opening it in AppMaker and working with the User Interface Editor as described in this chapter.

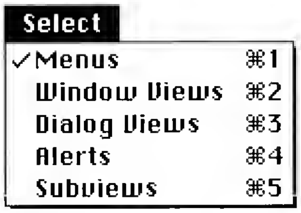
Introducing the User Interface Editor

Every Macintosh application contains **resources**, stored separately from the application's code, that form the application's user interface; these resources allow the user to control and interact with the application. AppMaker's User Interface Editor, the most visible component of AppMaker, provides a straightforward way to create and edit the resources that define the menus, windows, dialogs, alerts, and the items contained in them, of your application's user interface. For example, the editor provides tools for creating standard items such as buttons, check boxes, and static text, and for creating less common items such as palettes, pop-up menus, picture buttons, and custom sliders.

You can create resources for a new application; you can also create, change, and delete resources in an existing application, or copy resources from one application and paste them into another.

Components of the User Interface Editor



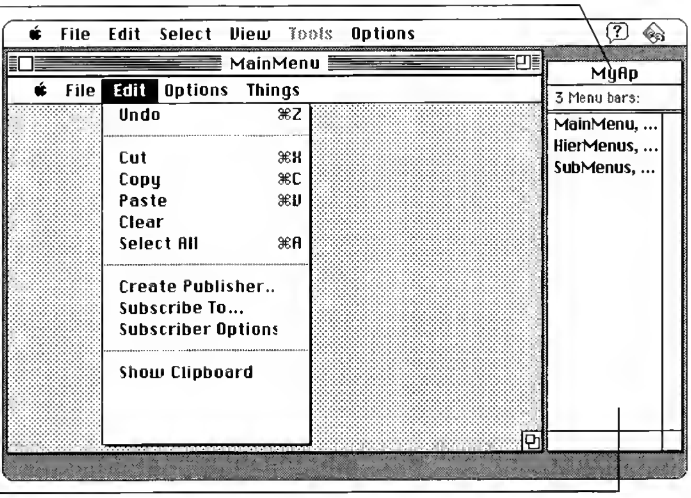


AppMaker's Select menu is the central control for the User Interface Editor. From the Select menu, you choose the category of resource you want to work with: Menus, Windows, Dialogs, or Alerts. (If you are using MacApp, the Select menu is slightly different, as shown.) Then AppMaker makes available its menus and tools for editing that category of resource. This chapter explains how to create and change every aspect of the user interface by working with these resources.

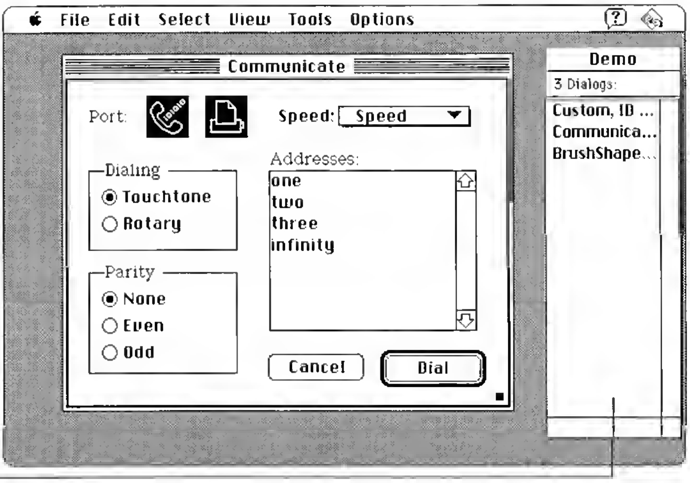
AppMaker's **List window**, always open when an AppMaker document is open, lists all the resources that are already defined for the selected resource category for the application you're working on. (AppMaker creates several standard resources for every new application, so even if you haven't created any resources, you'll see some in the List window for some resource categories. For a list of the standard resources that are created in new applications, see Appendix C, "Standard Resources.")

The List window is the first window you see when you create a new application or open an existing application. Initially, the Select menu has Menus checked, and the List window shows any menu bars that are defined.

When you select a menu bar from the List window (by clicking its name in the list), AppMaker displays the **Menu window**. This window contains a menu bar that looks and behaves very much like the real menu bar at the top of the screen, letting you create and edit all the menus for your application just by clicking and typing. In the figure below, Menus is chosen in the Select menu, the List window shows the standard menu bars included in every new application created with AppMaker, and the Edit menu is selected, showing the standard Edit menu AppMaker creates for you. (You can change this menu in any way that suits you, or delete it altogether.)



When you've chosen Windows, Dialogs, or Alerts from the Select menu (or Subviews if you're using MacApp), and clicked an existing resource in the List window (or created a new window, dialog, alert, or subview), AppMaker displays an **Items window** for the resource. The kind, position, size, and contents of the Items window depend on the category of resource you've selected or created. For example, if you create the most common kind of dialog box (DBox), AppMaker creates it with a standard border, a Cancel button, and an OK button which is the preset button. (Of course, you can change the standard items that AppMaker creates for you, along with all the other characteristics of the dialog box.)



Items window for the Communicate dialog

List window lists the Dialog resources when Dialogs is chosen from the Select menu

More About Resources

Every resource has a **type**, an **ID**, and a **name**. Some resources also have a **title**. When you create certain types of resources, such as menu bars, windows, dialogs, and alerts (and for MacApp, subviews), AppMaker will ask you to name the resource, and will assign an ID number, which you can change if necessary. This section covers general information about resources that will help you provide an appropriate name, and optionally your own ID number or title, when creating or changing a specific resource.

Resource types are defined by Apple Computer, and are distinguished by four-letter labels (such as DLOG for dialog, and WIND for window). AppMaker assigns these labels automatically based on the resources you create.

Resource IDs are numbers that uniquely identify each resource within its resource type. Apple Computer has reserved ID numbers up to 127 for its own use. IDs from 128 to 32767 are available for users.

Whenever you create a new resource, such as a new menu, AppMaker assigns to it the next available ID within the resource type. ("Next available ID" means the next unused ID after the most recently-created resource of the type. For example, if resources are created with IDs 201, 202, 203, then ID 203 is renumbered to 57, then the next resource of that type will have ID 58, if 58 is not already used.)

If you accept AppMaker's choice of ID, you'll be sure of not accidentally reusing an existing ID. However, you can change AppMaker's ID assignment if you need to, by typing a new ID in the dialog in which you provide information about the new resource.

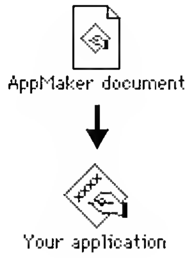
Note: ID numbers between 32768 and 65535 are converted to negative numbers from -32768 to -1. If you type an ID greater than 65535, AppMaker will truncate it to a valid ID.

When you create a resource, AppMaker requires that you give it a **resource name.** The name should be unique in the application, and should consist of up to 25 characters, which can include letters and digits as long as it doesn't begin with a digit (no punctuation, symbols, or spaces). AppMaker uses resource names extensively. First, they are used in the List window to show the existing resources of the kind checked in the Select menu. They are also used to match hierarchical menu items with their submenus, pop-up menu items with their menus, and menu items with the dialogs they invoke. In addition, resource names are used as filenames for the generated code and to form identifiers within the generated code. (For resources created outside of AppMaker without names, AppMaker uses the resource ID for these purposes instead of the name.)

A **title** is an optional piece of information that you can provide for a window or dialog resource. The title is more flexible than the resource name because it can contain any characters you can type, and can be as long as 255 characters. The title is displayed in the window's title bar when you are working in AppMaker. If you don't specify a title, AppMaker will make the title the same as the resource name.

Note: At run-time, for procedural languages and for the THINK Class Library, the title is also displayed in the title bar of an unnamed document (e.g., a new document that hasn't been saved). MacApp substitutes "Untitled-n" for the brackets (<<<>>>) in the Title field, and includes any other characters outside the brackets as typed. ("n" is an integer that increments with the number of unnamed documents opened, such as Untitled-1, Untitled-2, etc..) If the user has named the file, AppMaker-generated code for procedural languages and for the THINK Class Library substitutes the user's file name for the specified window title. MacApp substitutes the user's file name for the brackets in the Title field, and includes any other characters outside the brackets as typed.

Creating and Opening Applications

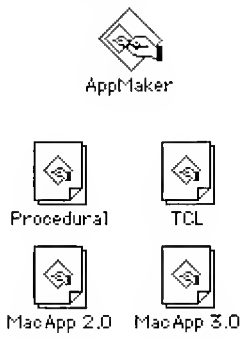


When you create a new application with AppMaker, you create an AppMaker document. In creating the document, you choose the programming language AppMaker should use to generate code for the user interface. (For information about choosing a language, see the next section, "Choosing a Programming Language.") After you design the user interface, generate source code, compile, and link, the AppMaker document becomes an application.

You can open and edit an AppMaker document, an application (created in AppMaker or elsewhere), a ResEdit resource file (file type 'rsrc'), or a ViewEdit resource file (file type 'SF20'). For convenience, we will refer to any of these as an application.

You can open several applications at the same time. This lets you copy and paste resources from one application to another.

To create a new application from the Finder:



1. Double-click the AppMaker application icon to run AppMaker.

   AppMaker displays a dialog box that shows the applications in the current folder.

   OR

   If you're using System 7.0 or later, you can double-click the stationery pad for the category of programming language you want to use for the application: Procedural, TCL (Object-oriented using THINK Class Library), MacApp 2.0, or MacApp 3.0 (Object-oriented using MacApp). (For information on choosing a language, see the next section, "Choosing a Programming Language.") AppMaker displays a dialog box asking you to name the new application. Skip to Step 3 below.

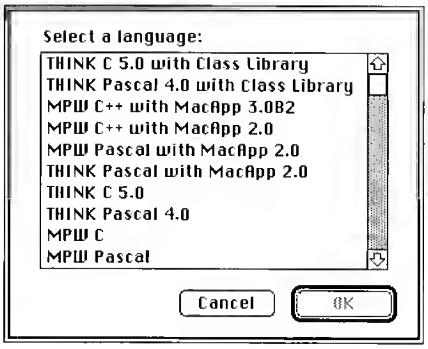
2. Click the New button in the dialog box.

   AppMaker displays a dialog box asking you to name the new application.

3. Type a name for the application, and optionally choose a different folder in which to store it, then click the Save button.

   Note: If you use MPW, you should not use a single quotation mark (') in the name. The generated .make file will not work properly if the name contains this character.

   AppMaker displays a dialog box asking you to select a programming language. (If you started by double-clicking a stationery document for the language category, you still need to pick the specific language that you want to use within the category. For example, if you've opened the TCL stationery document, you can choose either THINK C 5.0 with Class Library, or THINK Pascal 4.0 with Class Library.)



Select a language:
- THINK C 5.0 with Class Library
- THINK Pascal 4.0 with Class Library
- MPW C++ with MacApp 3.0B2
- MPW C++ with MacApp 2.0
- MPW Pascal with MacApp 2.0
- THINK Pascal with MacApp 2.0
- THINK C 5.0
- THINK Pascal 4.0
- MPW C
- MPW Pascal

4. Click a language, then click OK.

   AppMaker creates a new AppMaker document, copies several standard resources into it, and opens a List window with the title of your new application.

   Note: You can later change the language, but the new choice must be from the same category as the original choice, because the resources AppMaker supplies are different for each category of language. For more information, see the next section, "Choosing a Programming Language."

To create a new application from within AppMaker:

1. Choose New from the File menu.

   AppMaker displays a dialog box asking you to name the new application. Continue from Step 3 above.

   OR

If you're using System 7.0, you can open a stationery pad by choosing Open from the File menu, then selecting the stationery pad appropriate to the category of programming language you will use. After you click the New button in the dialog box, AppMaker displays a dialog box asking you to name the new application. Continue from Step 3 above.

To open an AppMaker document from the Finder:

AppMaker document

1. Double-click the AppMaker document icon.

   AppMaker opens a List window with the title of the AppMaker document.

   OR

   If you're using System 7, drag the icon for the document, application, or resource file you want to open to the AppMaker icon.

   Note: Double-clicking the icon of an application or resource file would run the application or ResEdit or ViewEdit, instead of running AppMaker. If you're using a System earlier than System 7, you can open existing applications or resource files only from within AppMaker, as described below.

To open an AppMaker document or an existing application from within AppMaker:

1. Choose Open from the File menu.

   AppMaker displays a dialog box that shows the applications in the current folder.

2. If necessary, change to a different disk or folder, then click the name of the file you want to open, then click the Open button.

   AppMaker opens a List window with the title of the application.

   Note: When you open an application that was not created with AppMaker, AppMaker looks for a language specifier. If there is no language specifier, or the language is one that AppMaker doesn't recognize, you'll see a dialog that asks you to select a programming language. To continue, select the language in which the application was programmed, then click OK.

Choosing a Programming Language

Because procedural languages and object-oriented languages require different kinds of resources, you need to choose the language before starting to design. After you select a language, AppMaker copies the appropriate standard resources into the new application. (For a list of the standard resources that AppMaker creates for each category of language, see Appendix C, "Standard Resources.")

AppMaker supports these categories and languages:

- ☐ Procedural (traditional) languages
  - ☐ THINK C
  - ☐ THINK Pascal
  - ☐ MPW C
  - ☐ MPW Pascal
  - ☐ A/UX ANSI C
  - ☐ FORTRAN*
- ☐ Object-oriented using THINK Class Library (TCL)
  - ☐ THINK C with Class Library
  - ☐ THINK Pascal with Class Library
- ☐ Object-oriented using MacApp 2.0
  - ☐ THINK Pascal with MacApp 2.0
  - ☐ MPW Pascal with MacApp 2.0
  - ☐ MPW C++ with MacApp 2.0
  - ☐ Modula-2 with MacApp 2.0
- ☐ Object-oriented using MacApp 3.0
  - ☐ MPW C++ with MacApp 3.0
- ☐ XVT*

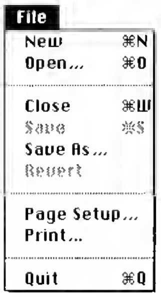
*FORTRAN and XVT are add-on products that are available separately from Bowers Development Corporation.

After you've created an AppMaker file, you can switch to a different language if needed (by using the Language command in the File menu); however, the new language must be from the same category as the original choice, or the generated code may not run because needed resources are missing. For example, if you switch from MPW Pascal (procedural) to THINK Pascal with Class Library (object-oriented), you will need to manually modify the code in order to make sure the appropriate resources and library routines are used.

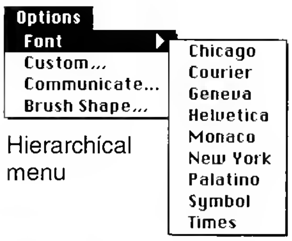
Working with Menus

To work with menus, you first choose Menus from the Select menu. The List window then lists any menu bars that are contained in your application, and you can select one to work with, or create a new one.

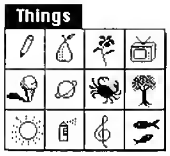
When you create a new application, AppMaker creates a standard main menu bar in the application, with standard Apple ( ), File, and Edit menus. For most applications, you'll want to add new menus and perhaps change the standard menus. (For example, AppMaker creates an Edit menu that includes the standard Publish and Subscribe commands for System 7 applications that support automatic updating of documents. If your application doesn't support these options, you'll want to delete them from the Edit menu.)



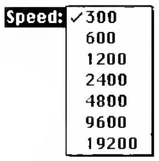
Pull-down menu



Hierarchical menu



Picture menu



Pop-up menu

With AppMaker's menu editor, you can create, change, and delete:

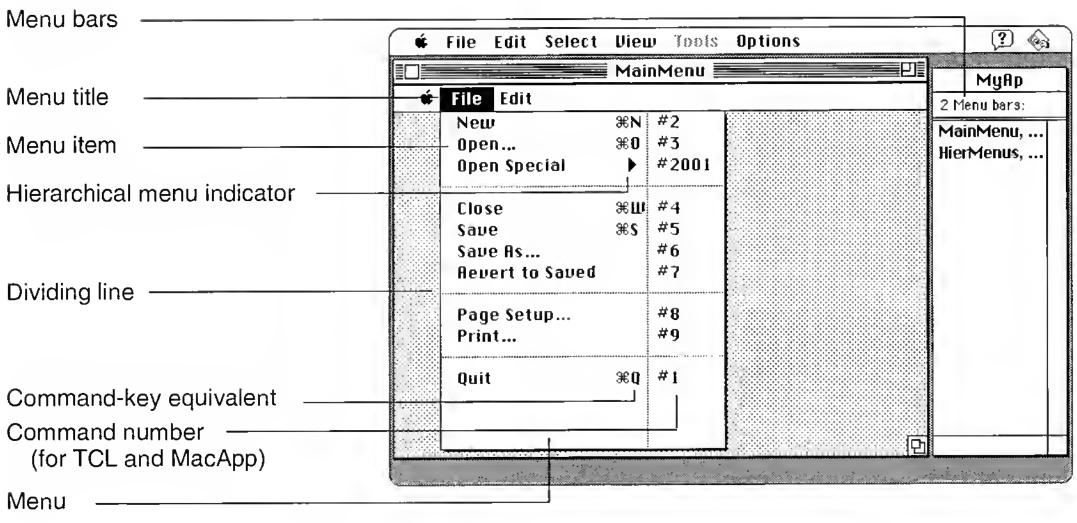
- Standard pull-down menus
- Hierarchical menus
- Picture menus (Can also be viewed as text)
- Pop-up menus

  Note: The contents of a pop-up menu are edited in the menu editor; the label and pop-up box are created as an item in the window or dialog where the pop-up is displayed. For more on creating a pop-up menu, see the sections "Basic Steps for Creating Items" and "Pop-up" in the later section, "Creating Items."

In working with menus, you can manipulate these elements:

- Menu bar
- Menu title
- Menu
- Menu item
- Submenu
- Command-key equivalent
- Hierarchical menu indicator
- Command numbers
- Dividing line
- Picture

  Note: The picture itself must be created in another application, and can be cut or copied into the AppMaker file to create a picture menu. (For more on creating picture menus, see the later section "To Create a Picture Menu.")



Creating Menus

You can add menus to an existing menu bar, or create a new menu bar and add menus to it.

You would create a new menu bar in order to have different menu bars in different modes. For example, HyperCard® has certain menus that appear in the menu bar only if a paint tool is selected. This can be achieved by having two menu bars.

Another reason for having more than one menu bar is to support hierarchical menus or pop-up menus; AppMaker places these in "HierMenus" and "Submenus" menu bars separate from the main menu bar. You can either create the submenu bar and its submenus before creating the hierarchical or pop-up menu, and reference the submenu you've created, or have AppMaker create the submenu bar and its submenu for you.

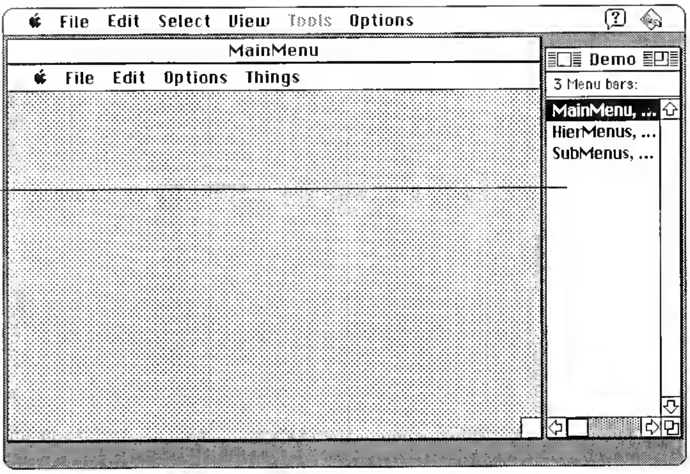
1. If necessary, choose Menus from the Select menu.

   In the List Window, AppMaker displays any menu bars already defined in the application.

2. Click a menu bar to work with.

AppMaker opens the Menu window with the selected menu bar. The List window remains the active window, allowing you to quickly examine several menu bars. As a shortcut, you can double-click a menu bar in the List window to open and activate the menu window.

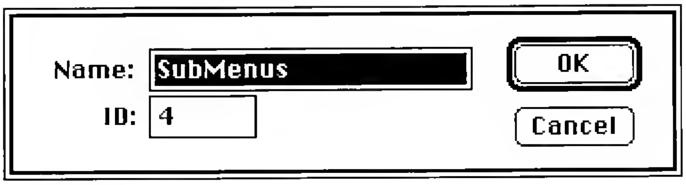
The List window stays active until you double-click a menu bar, or click its Menu window



OR

Create a new menu bar by choosing Create Menu Bar from the Edit menu.

AppMaker displays a dialog box asking for information about the new menu bar:


Name: SubMenus
ID: 4

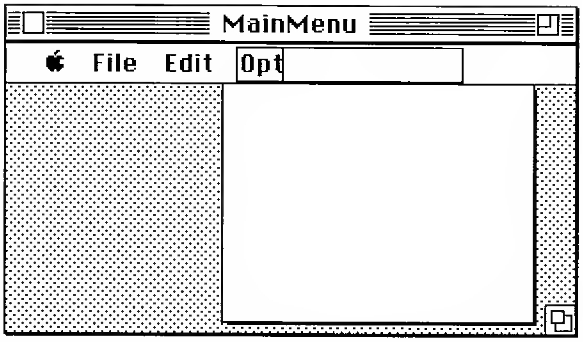
- ☐ The preset resource name for the new menu bar is SubMenus. To change the name, type a unique resource name of up to 25 characters composed of letters and/or digits in the Name field.
- ☐ If you want a different resource ID, press the Tab key to select the contents of the ID field, then type an ID.
- ☐ When the Name and ID are as you want them, click OK.

  AppMaker displays the Menu window with the resource name as its title, displaying an empty menu bar.

3. If necessary, activate the Menu window by clicking in it.

4. Choose Create Menu from the Edit menu.

   AppMaker displays an empty menu with an insertion point in the empty menu title. If the menu bar already contains menus, the new menu is added to the right of the existing menus. (You can change its position by dragging the menu's title where you want it, or by cutting and pasting the menu, as explained in the next section, "Changing Menus.")



AppMaker displays menus with a standard width and height. In the compiled application, the menu is as tall as the items in it, and as wide as the widest item. (To view the number of items in a menu in relationship to the 9-inch Classic and 13-inch Macintosh II screen sizes, choose Show Screen Sizes from the Options menu.)

5. Type a menu title, then press the Return key to type the first menu item.

   AppMaker creates a new menu with the resource name the same as the menu title, resource ID the next available ID, and menu ID the same as the resource ID. Remember, to avoid duplicate identifiers in the generated code, you should make all resource names in your application unique in the first 25 characters.

   Although submenu titles don't appear at run-time, you still should title them, because AppMaker uses the title to match a submenu with its associated hierarchical menu items or pop-up items, as well as to identify the submenu in the generated code.

6. Type the menu item, and optionally a Command-key equivalent after pressing the Tab key. (For the Command-key equivalent, AppMaker automatically converts a lower-case alphabetic character to upper case.)

   A menu item that brings up a dialog box should be followed by an ellipsis (...). You can type an ellipsis with three periods or with Option-semicolon (;).

   Tip: If you create a dialog box before creating the menu item that brings up that dialog and if you give them both the same name, AppMaker will remind you if you forget the ellipsis.

7. If you're using the THINK Class Library or MacApp, you should also specify a command number to all menu items (including hierarchical menu items) except divider lines:

   - ☐ Display the command number column if necessary, by choosing Show Command Numbers from the Options menu.

     AppMaker displays the command number column at the right edge of the menu. (Command numbers are not visible in the compiled application. This column is used by AppMaker to create command number resources.)

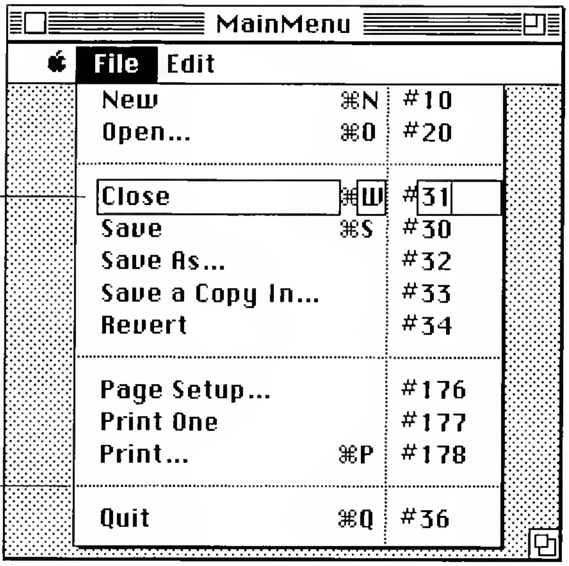
   - ☐ Press Tab to position the insertion point after the number sign (#) in the command number column, then type a unique command number. The number must be larger than the reserved numbers of 1 to 1023 if you are using the THINK Class Library, or 1 to 999 if you are using MacApp; it's best to use numbers starting with 2000 to avoid conflicts in cases where AppMaker needs to use command numbers starting at the low end of the available range.

8. Continue typing menu items, by pressing Return to create the next new item, then Tab to type a Command-key equivalent and, if necessary, a command number.

   You can add a dividing line (a gray line across the entire width of the menu) to group items on a menu, by typing a hyphen (-) as a menu item.

   To create a hierarchical menu, or a picture menu, see the next sections.

9. To finish adding menu items, press the Enter key or click outside of the menu.



To type a Command-key equivalent or command number, press Tab

To create a dividing line, type a hyphen (-) as the menu item

To add menu items to an existing menu:

1. If the menu is not currently selected, click its title to select it.

   AppMaker highlights the selected menu title and drops down its menu.

2. Click where you want to add the new menu item:

   □ To add the item at the end of the menu, click below the last menu item.

   □ To add an item above another item, click at the beginning of the existing item, then press the Return key.

   □ To add an item below another item, click anywhere in the existing menu item after the beginning, then press the Return key.

   AppMaker displays a field for typing the new menu item.

3. Type the item, its optional Command-key equivalent, and, if needed, its command number, as described in steps 6 through 9 above.

To create a hierarchical menu:

A hierarchical menu item references a separate menu (called a submenu), to be displayed when the user presses the hierarchical menu item. You can have AppMaker create the submenu for you when you create the hierarchical menu item. Or, you can create the submenu first and reference it when you create the hierarchical menu item.

1. Create a menu item as described above and type the command that will display the submenu.

   If the menu item you type is the same as the title of an already-created submenu, AppMaker automatically references the existing submenu. When you press Enter or click outside the menu, AppMaker displays the filled triangle indicating a hierarchical menu item, and the process of defining the hierarchical menu is complete.

   If you haven't yet created the submenu, or its title is different from the hierarchical menu item that references it, continue with the steps below.

2. Press the Tab key to select the field for the Command key equivalent.

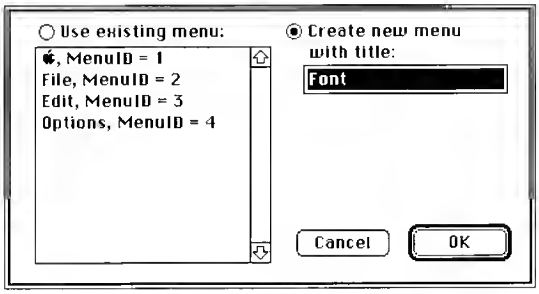
3. Choose Create Submenu from the Edit menu. Or use the keyboard shortcut, Command-K.

   AppMaker displays a dialog box in which you specify the menu to use for the submenu.

4. To use an existing menu, click a menu in the list on the left, then click OK.

   OR

To create a new menu for the submenu, optionally type a new menu title in the field on the right, then click OK. (If you don't type a new title, AppMaker titles the submenu the same as the hierarchical menu item.) If a SubMenus menu bar does not already exist, AppMaker creates it, then creates the new submenu in the SubMenus menu bar. Later, you can add menu items to it, as described in the earlier section "Creating Menus."



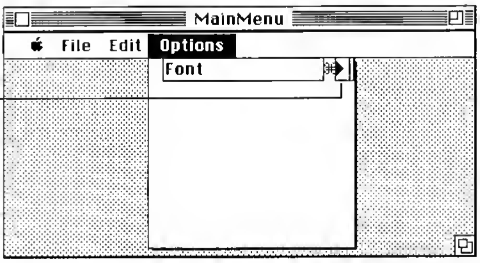
, MenuID = 1
File, MenuID = 2
Edit, MenuID = 3
Options, MenuID = 4

Font

Note: If you are using MacApp, any hierarchical menus must be in the menu bar with a resource ID of 130. AppMaker creates this menu bar, called HierMenus, when you create a new application using MacApp. When you use the Create Submenu command, AppMaker puts the new submenu in the HierMenus menu bar. If you create menus yourself that are to be used as hierarchical menus, you should create them in the HierMenus menu bar.

AppMaker displays the filled triangle that indicates a hierarchical menu item.

Filled triangle indicates a hierarchical menu item



5. Continue adding menu items, or, to finish defining this item, press the Enter key or click outside of the menu.

To create a picture menu:

Inside Macintosh describes how to build non-standard menus, for which the programmer must provide a menu definition procedure (MDEF). AppMaker provides a custom MDEF to implement picture menus. The Tools menu in AppMaker is a picture menu. Many other applications also provide examples of picture menus.

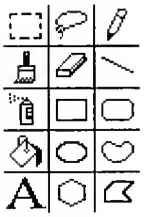
You can use a picture menu anywhere you can use a text menu: in the menu bar, as a hierarchical menu, or as a pop-up menu.

To create a picture menu, you first create a standard text menu. It is a good idea to provide as many text menu items as there are components of your picture menu. The text menu items provide meaningful names in the generated code and provide a place to specify Command-key equivalents and command numbers. They are also used internally to mark a selected item and by the Toolbox Menu Manager to blink a selected item.

After you've created both a text and a picture version of a menu, you can alternate between display of the text menu and the picture menu at any time by using the View menu. You can also change the menu from picture to text for the compiled application, as described in "To change a picture menu" in the later section "Changing Menus."

1. In a graphics application, create a picture in PICT format that exactly represents the way you want the picture menu to look. The picture should be divisible into a regular grid of picture components that will each be selectable as a picture menu item, and it should not have an outside border. (The Menu Manager will draw a shadowed border outside the picture.)

Create a picture composed of a regular grid without a border



2. Use ResEdit or an equivalent application to copy the picture into the application.

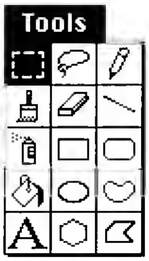
   OR

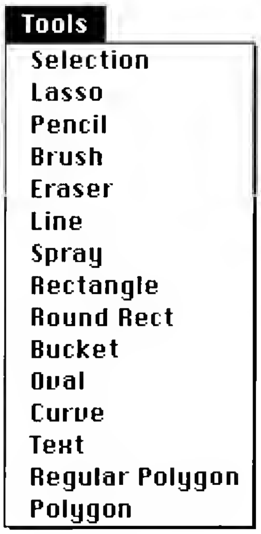
   Paste the picture from the Scrapbook or Clipboard into an Items window in the application. (For instructions on using the Scrapbook or Clipboard, see the reference manual that came with your Macintosh.) While the picture is still selected, press the Delete key or choose Cut or Clear from the Edit menu. The picture will be deleted from the Items window but will still exist in the application. (If you want to delete the picture from the application itself, you can do so by using ResEdit or equivalent.)

3. Create a text menu as described earlier in "Creating Menus." If the picture menu is to be a hierarchical or pop-up menu, create the menu in a Submenus menu bar.

   Create as many text menu items as there are picture menu items in the picture. The text items should correspond to the picture items in reading order (top-to-bottom and left-to-right).

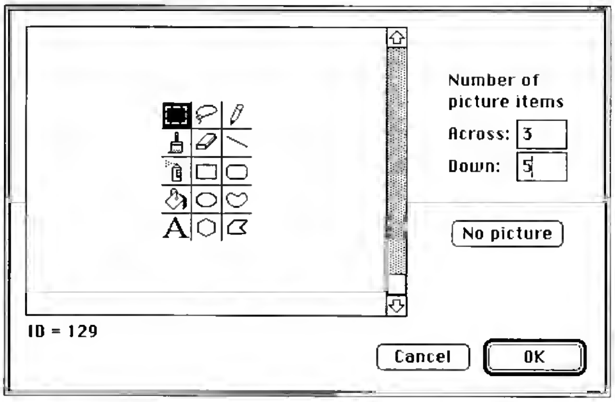
Text menu items correspond to picture menu items from left to right and top to bottom





4. Choose Menu as Picture from the View menu. Or, choose Menu Info from the View menu, then click Get Picture.

   AppMaker displays a dialog box asking for information about the picture menu.



5. Make sure the picture you want to use is displayed in the scrollable area. Use the scroll bar if needed to find the picture.

6. Type the number of picture menu items across and down into the Across and Down fields.

   AppMaker highlights the upper left corner of the picture to show what it will look like when the first picture menu item is selected. If the number of picture items is specified correctly, only the top left picture component should be highlighted.

7. Click OK.

   AppMaker displays the menu as a picture, with the first item selected. You can view the menu as text or as a picture by using the View menu.

Changing or Deleting Menus

You can change or delete any of the components of menus in AppMaker. There are four basic types of change:

- Changing or deleting the *contents* of menu bars, menu titles, menu items (including dividing lines), Command-key equivalents (including hierarchical menu indicators), or command numbers.
- Changing the *order* of menus or menu items.
- Changing the *resource name* or *resource ID* for a menu bar or menu. (You can also change the menu ID for a menu, although the menu ID is generally the same as the resource ID.)
- Changing a *picture menu* by viewing it as text, by changing the picture, or by deleting the picture so the menu is a text menu. (You change a text menu to a picture menu by the process described in the previous section, "To Create a Picture Menu.")

Of course, you can also add to menus at any time by following the instructions in the previous section "Creating Menus."

The basic procedure for changing menu information is as follows:

Basic Steps for Changing Menus

1. Choose Menus from the Select menu.
2. Select what you want to change or delete.

   If you want to change menu item text, Command-key equivalents, or command numbers for a picture menu, first choose Menu as Text from the View menu so you can have access to these characteristics.
3. Type the new information or delete the contents as needed, or to change other information about the menu bar or menu, choose Menu Bar Info or Menu Info, as appropriate, from the View menu.

To change the order of menus or menu items, drag a menu or menu item from its current location to the new location, or use the Cut and Paste commands in the Edit menu to move it.

Note: For dragging, a menu title should be highlighted, but not contain an insertion point, and a menu item should be unselected.

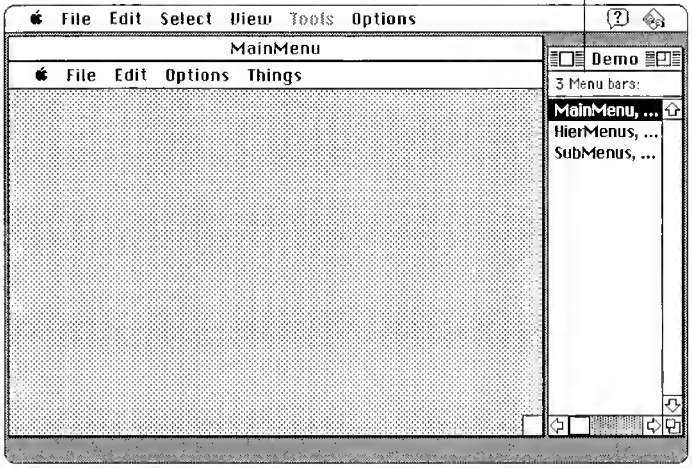
The next several sections go into more detail about how to make each of the kinds of changes and deletions.

To change or delete contents:

1. Choose Menus from the Select menu.
2. In the List window, select the menu bar you want to work with by clicking its name.
3. If you want to delete a menu bar, *which also deletes ALL the menus it contains*, choose Cut or Clear from the Edit menu.

   If the Menu window for the menu bar was open, AppMaker closes it and deletes the selected menu bar from the List window.

When you delete a menu bar, you are deleting ALL the menus it contains



OR

To change or delete menus, click in the Menu window to activate it. Then select the menu you want to work with by clicking its title.

AppMaker highlights the selected menu title and drops down its menu.

4. Make any of the following changes as needed:

   - ☐ To delete the entire menu, press the Delete key or the Clear key, or choose Cut or Clear from the Edit menu.
   - ☐ To change or delete a menu title, menu items (including dividing lines), Command-key equivalents (including hierarchical menu indicators), or command numbers, click or drag to select the information you want to change or delete in the selected menu. (To edit or delete command numbers, make sure they are displayed; i.e., Show Command Numbers is checked in the Options menu.)

     Use the standard text editing techniques to edit or delete characters. To delete a menu item, delete all of its characters.

   Note: To change or delete textual information for a picture menu, first display the picture menu as text, by selecting the menu, then choosing Menu as Text from the View menu. To change other aspects of a picture menu, see the later section "To Change a Picture Menu."

To change the order of menus or menu items:

You can change the order of menus in the menu bar, or change the order of menu items within any menu, by dragging.

You can also move a menu from one menu bar to another, or move menu items from one menu to another (in the same or a different menu bar), by using the Cut and Paste commands in the Edit menu to relocate the selected object.

1. Select the menu you want to work with by clicking its title.

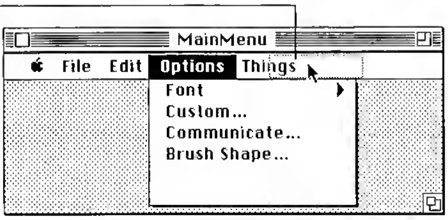
   AppMaker highlights the selected menu title and drops down its menu.

2. To move the entire menu, drag the highlighted menu title to the new location in the menu bar. When you release the mouse button, AppMaker redraws the menu in its new location.

OR

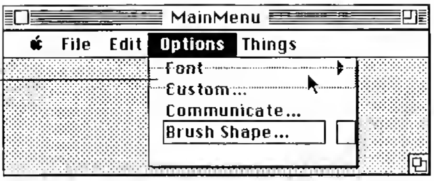
To move the menu to a different menu bar, choose Cut from the Edit menu. Then display the other menu bar (by clicking its name in the List window, or creating a new menu bar), and choose Paste Menu from the Edit menu. AppMaker pastes the menu at the right end of the menu bar, and you can drag it to a new location if needed.

Drag a menu to a new location in the menu bar



3. To move a menu item, drag the unselected item up or down to its new location in the menu. AppMaker moves the menu item and any associated Command-key equivalent, hierarchical menu indicator, and command number to the new location.

Drag a menu item to a new location in the menu



OR

To move a menu item to a different menu (in either the same menu bar or a different menu bar), in turn cut, then paste in the new location any associated Command-key equivalent, hierarchical menu indicator, and command number that belong with the menu item, then cut and paste the text of the menu item itself. (When you cut the text of a menu item, the entire item is deleted.)

For instructions on how to add a new blank item to a menu so you can paste the components of the menu item in their proper places, see the earlier section "To Add Menu Items to an Existing Menu."

To change the resource name or resource ID for a menu bar or a menu (or menu ID for a menu):

1. Select the menu bar or the menu you want to work with:

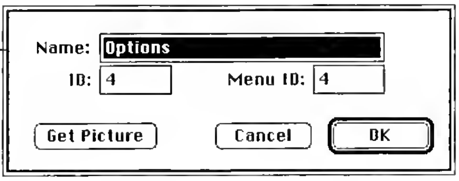
   ☐ Select a menu bar by clicking its name in the List window.

   ☐ Select a menu by clicking to activate the Menu window, then clicking the title of the menu.

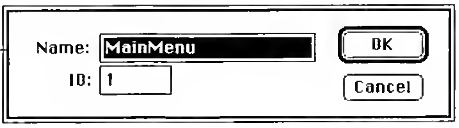
2. Choose Menu Info or Menu Bar Info from the View menu, according to what you have selected.

   AppMaker displays the Info dialog box for the selected menu or menu bar.

Info for a menu


Name: Options
ID: 4
Menu ID: 4

Info for a menu bar


Name: MainMenu
ID: 1

   For a menu bar, this dialog box is identical to the one used to create the menu bar, so the resource name and ID are the same as those you accepted or edited in creating the menu bar. For a menu, the preset resource name is the menu's title, the resource ID is the next available ID, and the preset menu ID is the same as the resource ID.

3. Change the resource name or resource ID by editing the fields for Name and ID. Change the menu ID by editing the field for Menu ID.

   Remember that resource names should consist of only letters and or numbers, up to 25 characters, and should be unique within the application. The resource ID should be a number between 128 and 32767, and should be unique within the resource type.

   Note: Keep in mind that the Menu ID should be the same as the resource ID unless you have a specific purpose in making them different, and are prepared to handle the consequences.

4. When you are done making changes in the dialog box, click OK.

To change a picture menu:

After you've created a picture menu (as described in the earlier section "To Create a Picture Menu"), you can view the picture menu in its textual form, change the picture used for the menu, or convert the picture menu to a text menu if that's how you want it to appear in your final application. (Viewing a picture menu as text allows you to change menu item names, Command-key equivalents, and command numbers.)

1. If the picture menu is not currently selected, click in its menu title to select it.

   AppMaker highlights the selected menu title and drops down its menu.

2. Make the changes as follows:

   - To change the view of the menu, choose Menu as Text from the View menu. AppMaker displays the picture menu in its textual form. You can view it in its picture form again by choosing Menu as Picture from the View menu.
   - To change the picture used for the menu, choose Menu Info from the View menu, then click the Get Picture button. In the dialog box (which is the same dialog used for creating the picture menu), scroll to display the new picture, type the number of picture menu items across and down, and click OK. (For information on getting pictures into the application, see the earlier section "To Create a Picture Menu" in "Creating Menus.")
   - To convert the picture menu to a text menu, removing the picture, choose Menu Info from the View menu, then click the Get Picture button. In the dialog box (which is the same dialog used for creating the picture menu), click the No Picture button, then click OK. The menu will be a standard text menu in your application.

Working with Windows, Dialogs, and Alerts

Inside the Macintosh toolbox, **dialog boxes** and **alert boxes** are just special kinds of **windows**. So it should come as no surprise that AppMaker handles all three in the same way. For convenience, we can refer to them generically as windows but the remaining discussion applies equally to dialogs and alerts.

Note: For MacApp, AppMaker provides an additional view resource called a **subview**. A subview can't be used by itself because it doesn't have its own window; it must be included in a window view or a dialog view, either by programming it to load at run-time, or by using AppMaker's Include View tool to place the subview in an Items window (as described later in the section "Creating Items"). A subview can provide an alternate, or subset, view of a window or dialog, such as for describing the printable content of a window, or for specifying an alternate part of a dialog to be displayed when the user performs a particular action such as clicking a button. Creating and using subviews is similar to working with windows (AppMaker creates a window in order to display the subview), so information on subviews is included in this section.

The following table shows the types of windows and dialogs you can create, and some of their attributes:

Window Type	Title Bar (Drag Region)	Size Box (Grow Region)	Zoom Box (Zoom Region)	Optional Close Box (Go-away Region)	Scroll Bars
ZoomDoc	✓	✓	✓	✓	✓
NoGrowDoc	✓			✓	
ZoomNoGrow	✓		✓	✓	
Document	✓	✓		✓	✓
RDoc	✓			✓	
Movable	✓				
DBox					
PlainDBox					
AltDBox					
Other	Determined by programmer				

You select, create, and delete windows from the **List window**. (The type of window resource displayed in the List window depends on what you have chosen from the Select menu: Windows, Dialogs, or Alerts, or, in the case of MacApp: Window Views, Dialog Views, Alerts, or Subviews.)

When you select a window from the List window, AppMaker displays the window as it will be seen at run-time. It has the same type, position, size and other attributes that it will have at run-time. The items associated with the window—the buttons, static text, etc.—are displayed in the window. This displayed window is called an **Items window** because it displays a group of items for the associated window resource.

When you select (by clicking) a window from the List window, AppMaker displays the Items window behind the List window. The List window remains the active window, and, if an Items window is already displayed, it is first closed before displaying the new Items window. This is convenient for quickly examining several windows.

To work in the Items window, you must first activate it by clicking in it. As a shortcut, you can double-click in the List window to select an Items window and make it the active window.

The rest of this section describes how to create windows, dialogs, and alerts; how to change their resource name, ID, and other information; how to move and resize them; and how to delete them. The subsequent section, "Working with Items", describes how to create, change, and delete items in windows, dialogs, and alerts.

Creating Windows, Dialogs, and Alerts

You create a window or dialog box by specifying the type of window or dialog box you want, and supplying a resource name. Optionally you can supply a title, and, if necessary, change the resource ID that AppMaker assigns. You can also choose whether it will have a close box, and, if you are creating a window or modeless dialog box, whether it is visible at startup (shown automatically when your application opens a document), or visible only under conditions specified in the code.

AppMaker creates a resource describing the type of window you specify, and opens it, so you can change aspects of the window, such as size or shape, and add items to the window, such as buttons, scrollable lists, text, and so on. Depending on the type of window you create, AppMaker puts certain standard items in the window for you, which you can change or delete if needed.

Creating an alert is similar, but you don't need to specify a type of window, because alerts have one standard window style, and no title is requested because alerts don't have title bars. Alerts are visible only when conditions specified in the code require the alert to appear. (Also, the System automatically displays the appropriate icon in the standard position in an alert when your code calls one of the library routines, Acknowledge or Confirm, or the Toolbox routines, StopAlert, NoteAlert, or CautionAlert. For details, see the section “Miscellany Routines” in Chapter 5, “Programming a Procedural Application.”)

You can see how any window relates to the Macintosh 9-inch Classic and 13-inch Macintosh II screen sizes by choosing Show Screen Sizes from the Options menu. The 9-inch screen is 342 x 512 pixels and the 13-inch screen is 480 x 640 pixels.

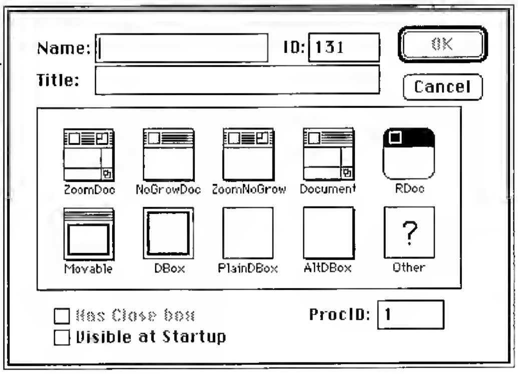
Note: When you create a subview in MacApp, you specify only a resource name and resource ID, not a window style or other characteristics, because the window AppMaker displays for the subview is only for the purpose of display and manipulation of items; the subview does not have its own window.

To create windows:

1. Choose what you want to create from the Select menu (Windows, Dialogs, or Alerts, or, for MacApp: Window Views, Dialog Views, Alerts, or Subviews).
2. From the Edit menu, choose Create Window, Create Dialog, or Create Alert, as appropriate. (For MacApp, there is the additional option of Create Subview.)

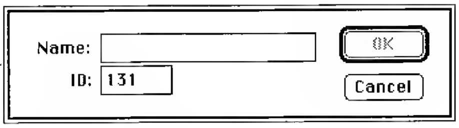
AppMaker displays a dialog box asking you to provide information about the new window. In the dialog box for creating windows or dialogs, the preset window or dialog type is shown with a black outline.

Dialog box for creating a window or dialog


Name: | ID: 131 | OK
Title: | Cancel

☐ Has Close box
☐ Visible at Startup
ProcID: 1

Dialog box for creating an alert or subview


Name: | OK
ID: 131 | Cancel

3. Type a resource name in the Name field, and, if you want to change the resource ID, press the Tab key, then type a new ID. (The resource name is restricted to 25 letters and digits, and the ID must be a number between 128 and 32767.)

   If you're creating an alert or a subview, that's all there is to it; you can click OK to have AppMaker to create the alert and open and activate the Items window. If you're creating a window or dialog, continue with the steps below.

4. If you want to specify a window or dialog title that is different from the resource name, press the Tab key, then type a title.

   The title can be any characters you can type up to 255 characters. If you leave the title empty, AppMaker makes it the same as the name.

Note: The title is used in AppMaker for display in the window's title bar, if any. At run-time, for procedural languages and for the THINK Class Library, the title is also displayed in the title bar of an unnamed document (e.g., a new document that hasn't been saved). MacApp substitutes "Untitled-n" for the brackets (<<<>>>) in the Title field, and includes any other characters outside the brackets as typed. ("n" is an integer that increments with the number of unnamed documents opened, such as Untitled-1, Untitled-2, etc..) If the user has named the file, AppMaker-generated code for procedural languages and for the THINK Class Library substitutes the user's file name for the specified window title. MacApp substitutes the user's file name for the brackets in the Title field, and includes any other characters outside the brackets as typed.

5. Select the window or dialog type you want to create, by clicking its picture in the dialog box.

   If you choose Other to use a programmer-defined window, you should also supply the ProcID, which is the resource ID multiplied by 16. The Other window type must be defined by a window definition function (WDEF resource) that you have copied into AppMaker using ResEdit or its equivalent. Both the THINK Class Library and MacApp support floating windows (remaining in front of all the other windows): the THINK Class Library supplies a WDEF with resource ID 200 and a ProcID of 3200; MacApp supplies a WDEF with resource ID 3 and a ProcID of 48.

   Note: When generating code for a dialog, AppMaker treats the four window types, Movable, DBox, PlainDBox, and AltDBox, as modal dialogs. It treats the remaining predefined types as modeless dialogs. (The "Other" window type is programmer-defined.)

6. As needed, click the check boxes for Has Close Box, and Visible at Startup.

   If Visible at Startup is not checked, a window will be visible only when your code makes it visible; a dialog box will be visible only when it is invoked by a menu command.

7. Click OK to create the window or dialog.

   AppMaker displays an Items window and selects it so you can immediately create items in the new window or dialog. See "Creating Items" later in this chapter.

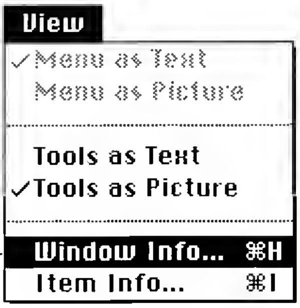
Changing or Deleting a Window, Dialog, or Alert

You can change any of the information you specified when you created a window, dialog, or alert, such as its resource name or the type of window. (For a subview, you can change the resource name or resource ID.) You can also move it, reshape it, or delete it along with all of its contents. (To add items to a window, dialog, or alert, see the later section, "Creating Items.")

When moving or reshaping a window, you can view changes in relation to the 9-inch Macintosh Classic and 13-inch Macintosh II screens by using the Show Screen Sizes command in the Options menu.

To change the name, ID, or other information for a window, dialog, or alert:

1. Choose Windows, Dialogs, or Alerts, as appropriate, from the Select menu. (If using MacApp, you can also choose Subviews.)
2. In the List window, click a window to select it.
3. From the View menu, choose Window Info, Dialog Info, or Alert Info, as appropriate. (If you chose Subviews from the Select menu (step 1), you can choose Subview Info from the View menu.)



The menu command changes to Dialog Info, Alert Info, or Subview Info depending on what type of Items window is displayed

AppMaker displays the same dialog box as when you created the window, dialog, or alert.

4. Change any of the information, then click OK.

   AppMaker redisplays the Items window with the new title or window type.

To move a window, dialog, or alert:

You move a window by using the standard Macintosh technique: you drag its frame. Even if a window doesn't have a title bar (for example, modal dialogs and alerts) it still has a narrow frame.

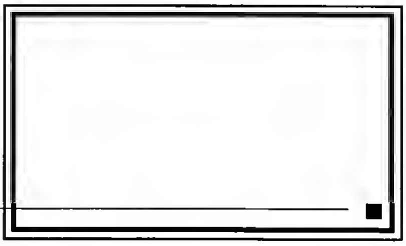
1. If the Items window is not already selected, select it by clicking in it.

2. Drag the window's frame to where you want it.

   When you release the mouse button, AppMaker redraws the Items window in its new position.

To reshape a window, dialog, or alert:

You reshape a window by using the standard Macintosh technique of dragging its lower-right corner. Even if a window doesn't have a size box (for example, modal dialogs and alerts) you can still drag the lower-right corner.



1. If the Items window is not already selected, select it by clicking in it.

2. Drag the lower-right corner where you want it.

   When you release the mouse button, AppMaker redraws the Items window with its new shape.

   If the window has scroll bars on its edges, AppMaker automatically moves and resizes them.

To delete a window, dialog, or alert:

Note: Deleting a window, dialog, or alert also deletes all of its contents.

1. Choose Windows, Dialogs, or Alerts, as appropriate, from the Select menu. (If using MacApp, you can also choose Subviews.)
2. In the List window, click to select the resource you want to delete.
3. From the Edit menu, choose Clear or Cut.

   AppMaker closes the Items window, and deletes the window, dialog, or alert with all of its contents.

Working with Items

With AppMaker, you can quickly create any of the following standard items:

- Buttons
- Check boxes
- Radio buttons
- Static text
- Edit text
- Gray lines
- Rectangles
- Labeled groups (called Clusters in MacApp)
- Pop-up menus
- Scrollable lists
- Scroll bars
- Icons
- Pictures

In addition, AppMaker makes it easy to create these less-common items:

- Tool palettes
- Picture buttons
- Picture check boxes
- Picture radio buttons
- Multi-picture buttons
- Sliders

AppMaker also gives you the ability to create, in effect, custom items, through the following:

- Controls defined by resources
- User items (programmer-defined items)

If you're using an object-oriented programming language with the THINK Class Library or MacApp, you have access to additional items, identified here with names based on their class names:

For TCL:

- CArrayPane
- CArrowPopupPane
- CDialogText
- CIntegerText
- CPanorama
- CRadioGroupPane
- CScrollPane
- CStyleText

For MacApp:

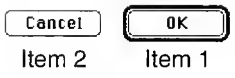
- TDialogView
- TGridView
- TNumberText
- TPattern
- TScroller
- TTextGridView
- TTEView
- Include View

As described in *Inside Macintosh*, dialogs and alerts have associated item lists which define their contents. The Macintosh toolbox does not provide a similar level of support for windows; they are effectively blank canvases. AppMaker extends the concept of an item list to windows. (Technical note: In procedural languages, a window item list is stored in a special AppMaker 'Witl' resource with the same ID as the window. This resource has the same structure as a 'DITL' resource. For languages using the TCL, AppMaker creates a 'WIND' and a 'Pan#' resource with associated pane resources; for MacApp, AppMaker creates a 'View' resource.)

Any kind of item can appear in a window, a dialog (modal or modeless), or an alert. (Of course, most items are not used in alerts. Static text and one or two buttons are usually all that's required. The appropriate icon in the standard position is automatically displayed by the System when your code calls the appropriate library or toolbox routine; you don't create the icon as an item.)

This section on working with items first explains some necessary background information about items, then tells you how to create items, select and change them, and delete them. These techniques are the same for any Items window, i.e., any window, dialog, or alert (or MacApp subview).

Order and Overlapping of Items

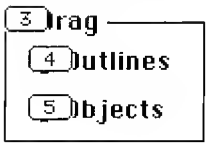


Within an item list, items are ordered by number. The item created first is item number 1 and so on. In modal dialogs and in alerts, items 1 and 2 have special significance: item 1 should be the OK button (or the default button) and item 2 should be the Cancel button.

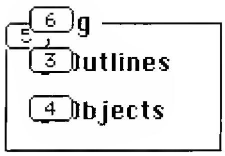
You can see the item number of a selected item by choosing Item Info from the View menu, or see the item numbers of all items in an Items window by choosing Show Item Numbers from the Options menu.

When you "nest" an item inside another item, AppMaker will renumber items if necessary to reflect their position in the nesting hierarchy, so they will be drawn and respond to mouse clicks in the correct order at run-time. (An item is inside another item if its upper-left corner is inside the "parent" or enclosing item.) For example, if you create a rectangle or a labeled group, then create radio buttons inside it, AppMaker will renumber items so that the radio buttons are drawn in front of the rectangle or labeled group that encloses them. (The numbering is different for procedural languages than for the TCL or MacApp.) Nested items are drawn right after their parent, before other items.

For TCL and MacApp, the nested items have higher numbers



For procedural languages, AppMaker renumbers the nested items to have lower numbers. (The group's title has a higher number because it is a separate static text item drawn after the rectangle.)



Note: Nested items must actually be *within*, not behind, the parent item to function properly. If you create some radio buttons, then draw a rectangle around them, the radio buttons will be behind the rectangle, and displayed as gray. Instead, create the parent item first, then its nested items inside of it, or drag items into the parent item, or correct the order by dragging the parent out of the way so it doesn't cover the items you want nested, then drag the items into the parent and reposition the parent as needed. (Technical note: For procedural languages, AppMaker infers the nesting order from the item coordinates in the resource; if some items have coordinates that are contained within another item, AppMaker considers the contained items to be nested. For the THINK Class Library and MacApp, the resource specifies the nesting order.)

Inside AppMaker, when items overlap, a mouse click in the overlap area is handled by the frontmost item (the one that appears visually to be in front of another item). Nested items are considered to be in front of their parent item, to be consistent with the behavior just described; a mouse click in the nested item is handled by the nested item, not by the parent.

You can renumber items by using the Cut and Paste commands in the Edit menu. When you cut (or clear) items, the remaining items are renumbered consecutively. When you paste items, they are numbered higher than the existing items provided they are not nested. Changing the nesting hierarchy of items, such as by moving them into or out of an enclosing item, also causes items to be renumbered.

Item Identifiers

For buttons, check boxes, radio buttons, static text, and edit text, AppMaker uses the item's title to form an identifier in the generated code. The title is the text you type to identify the item; for example, "OK" is the title for an OK button, the label next to a radio button is its title. To avoid duplicate identifiers in the generated code for these items, you should make the item titles unique in the first 25 characters. Items in dialogs should have titles that are unique within the dialog; items in windows should have titles that are unique in the application. (If you need to cheat on this, you can make a title unique by pressing Return, then typing more characters to distinguish it from existing titles; then reshape the item so the additional characters are not visible.)

For other items, AppMaker forms an identifier from the item's kind and number. Because the item number is unique within a dialog, this assures unique identifiers within a dialog.

Enabled or Disabled Status of Items

Every item has an enabled/disabled status. Put simply, an enabled item responds to mouse clicks, a disabled item does not. (For details, see *Inside Macintosh*, the section "Item Types" in the chapter "Dialog Manager.") When you create an item, AppMaker enables it or disables it depending on the kind of item. For example, buttons and check boxes normally respond to mouse clicks, so AppMaker creates them as enabled; static text and lines normally don't respond to mouse clicks, so AppMaker creates them as disabled. The following table shows the initial status of each kind of item:

Initial enabled/disabled status of items

Enabled	Disabled
Button	Static text
Check box	Icon
Radio button	Picture
Edit text	Line
Palette	Rectangle
Pop-up menu	
List	
Scroll bar	
Labeled group	
Scroller	
Custom control	
User item	

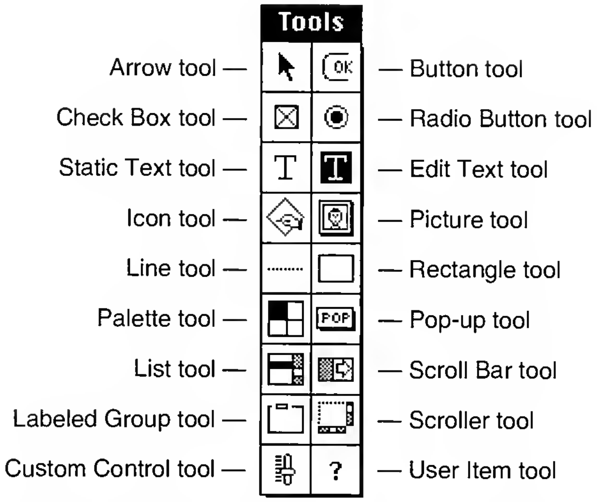
(All additional items that can be created using the THINK Class Library or MacApp are created as enabled.)

You can reverse the status of an item by holding down the Option key when you create it. Disabled buttons, check boxes, and radio buttons are displayed dimmed and AppMaker generates code to enable/disable them at run-time. (You must write code to set the conditions under which the item is to be enabled or disabled.) Enabled icons (available only for procedural dialogs) are displayed inverted and generated code treats them like radio buttons; i.e., the clicked icon in a group is highlighted and the other icons are dimmed.

You can change the enabled or disabled status of an item after you have created it by selecting the item, then choosing Item Info from the View menu. In the Item Info dialog, click the Enabled or the Disabled radio button to enable or disable the item, respectively.

Tools for Working with Items

In order to work with items, you use tools in AppMaker's Tools menu, which is available when the Items window is active. At the top left of the menu is the Arrow tool, which is used for selecting items in order to move or reshape them, change various characteristics of items, or delete them. The other tools in the menu are for creating each kind of item.



⌘-\

You can alternate between the Arrow tool and the currently selected tool by pressing Command-\ (back slash).

Viewing Tools as Text

If you prefer, you can view text labels for the tools in the Tools menu, instead of pictures. The tools will be listed in alphabetical order.

Viewing tools as text also allows access to additional tools if you are using MacApp or the THINK Class Library.

1. Activate an Items window:

   - ☐ Choose Windows, Dialogs, or Alerts from the Select menu.
   - ☐ In the List window, click the window you want to work with, then click the Items window to activate it. Or, double-click the window in the list to both open and activate the Items window.

     Alternatively, create a new window by choosing Create Window/Dialog/Alert from the Edit menu.

   The Tools menu is now available.

2. Choose Tools as Text from the View menu.

   AppMaker displays the Tools menu as a text menu. If you are using MacApp or the THINK Class Library, the text version of the Tools menu includes additional tools, as shown in the table below. The text label for the tool is its class name in MacApp and TCL.

3. When you want to see the Tools menu as pictures, choose Tools as Pictures from the View menu.

The following table shows the names of the tools for each language category. At the bottom of the table you can see the additional tools that are accessible for TCL and MacApp when you view tools as text.

Tool names for each language category

Procedural	TCL	MacApp
Arrow	Arrow	Arrow
Button	CButton	TButton
Check Box	CCheckBox	TCheckBox
Edit Text	CEditText (windows) CDialogText (dialogs)	TEditText
Icon	CIconPane	TIcon
Labeled Group	CLabeledGroup	TCluster
Line	CGrayLine	TGrayLine
List	CArrayPane	TTextListView
Palette	CPalette	TPalette
Picture	CPicture	TPicture
Pop-up Menu	CStdPopupPane	TPopup
Radio Button	CRadioControl	TRadio

Procedural	TCL	MacApp
Rectangle	CAMBorder	TControl
Scroll Bar	CScrollBar	TScrollBar
Scroller	CScrollPane	TScroller
Static Text	CStaticText	TStaticText
User Item	CPane	TView
Custom Control	Custom Control	Custom Control
	CArrowPopupPane	Include View
	CIntegerText	TDialogView
	CPanorama	TGridView
	CRadioGroupPane	TNumberText
	CStyleText	TPattern
	CTable	TTEView
		TTextGridView

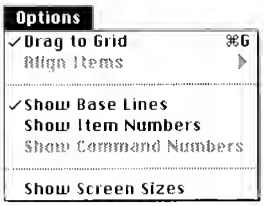
Additional tools for TCL and MacApp when viewing Tools as Text

Note: The Scroller tool is not available for procedural languages, but is available for TCL and MacApp when you view tools as pictures as well as when you view tools as text.

Creating Items

By using the Tools menu described in the previous section, you can create just about any interface item you need in a window, dialog box, or alert. This section describes the general procedure for creating items, then lists each item, with additional details about creating that particular type of item.

Before you begin creating items, decide whether you want the alignment help provided by AppMaker's invisible grid which is spaced at 4-pixel intervals. If the grid is on, then when you create or move an item its upper-left corner is positioned at the nearest grid intersection. When you reshape an item, its height and width are made a multiple of the grid size.



Use the Drag to Grid command in the Options menu to turn the grid off and on. (If you create items with the grid off, you can align them later by using the Align Items command in the Options menu, as described in the later section "Changing or Deleting Items.")

Basic Steps for Creating Items

1. If the item you will create uses a picture (for a picture, palette, picture pop-up menu, or custom control) create the picture(s) beforehand with another application and save them in PICT format in the size required. If you're creating an icon, the picture must be in ICON format (which can be created by ResEdit, among other applications). Then, to have the picture or icon available in your AppMaker document, either copy the PICT or ICON resource into AppMaker using ResEdit or equivalent, or copy the picture or icon into the Scrapbook and paste it into AppMaker as described below:

   - ☐ In the Scrapbook, scroll to the picture or icon, then choose Copy from the Edit menu.
   - ☐ Activate an Items window in the AppMaker document by clicking in it.
   - ☐ Choose Paste Picture or Paste Icon from the Edit menu. AppMaker will paste the picture or icon into the upper-left corner of the Items window. In the process, AppMaker creates the appropriate resource, assigns an ID, and creates a picture item or icon item as appropriate.
   - ☐ If you want the picture item or icon item in the window, use the Arrow tool to move it to where you want it.
   - ☐ If you want to use the picture or icon only to create other items, delete the picture item or icon item that was pasted. The PICT or ICON resource will still be in the AppMaker document, and will appear in AppMaker dialogs for creating various items, as described later in this section.

   Note: If you're creating a TPattern in MacApp, you must create a PAT resource in another application such as ResEdit, and copy it into AppMaker using ResEdit or equivalent. (It cannot be copied in via the Clipboard or Scrapbook.)

2. Activate the Items window in which you want to create items:

   □ Choose Windows, Dialogs, or Alerts from the Select menu. (If using MacApp, you can also choose Subviews.)

   □ To create items in an existing window, dialog, or alert, click the window you want to work with in the List window, then click the Items window to activate it. Or, double-click the window in the list to both open and activate the Items window.

   OR

   To create items in a new window, choose Create Window, Create Dialog, Create Alert, or Create Subview as appropriate from the Edit menu.

   The Tools menu is now available.

3. From the Tools menu, select the tool for the item you want to create.

   If you prefer to choose from a text menu, rather than pictures, choose Tools as Text from the View menu, then select a tool.

   The pointer changes to a crosshair or another shape suitable for creating the item.

4. Click in the Items window where you want the upper left corner of the item to be, or drag to establish the size of the item, starting at the upper left corner. (The initial size of an item needn't be exact, as you can change it at any time.)

   The table after these steps summarizes items by how they are created (either click and type a title, click and drag, or click and complete one or more dialog boxes).

   To reverse the initial enabled/disabled status for an item, hold down the Option key when you click or drag. For details on enabled/disabled status, see the earlier section "Enabled or Disabled Status of Items" in the introduction to "Working with Items."

5. If necessary, type a title for the item, or complete the dialog box that appears.

   For items that have titles, AppMaker displays a blinking insertion point so you can type the title. If the item is a check box, radio button, or static text, AppMaker automatically sizes the item to fit the text you type.

6. To create another item of the same type, click or drag where you want the next item.

   OR

   To create a different type of item, select its tool from the Tools menu, and click or drag to create it as described in the steps above.

   OR

   To move, change, or delete the item, select the Arrow tool from the Tools menu (or press Command-\ to switch to the Arrow tool), then follow the instructions in the later section, "Changing or Deleting Items."

   You can also press Enter to finish creating an item.

   Note: Newly-created items are selected to make it easy for you to move, change, or delete them. If an item is nested (such as items that you create inside a labeled group in the THINK Class Library, or a cluster in MacApp), AppMaker displays a bounding rectangle for any part of the selected item that extends beyond the edge of its parent, so you can see the size of the entire item. If you're editing the text of an item, all the text is shown. When not selected, a nested item is drawn clipped to the borders of its parent.

Summary of how items are created

	Click and type	Drag to size	Click and complete dialog
	Button	Edit text	Icon
	Check box	Line	Picture
	Radio button	Rectangle	Palette
	Static text	List	Pop-up
		Scroll bar	Custom control
		User item	
		Labeled group	
		Scroller	
Additional tools for TCL when viewing Tools as Text		CIntegerText	CArrowPopupPane
		CPanorama	
		CRadioGroupPane	
		CStyleText	
		CTable	
Additional tools for MacApp when viewing Tools as Text		TDialogView	Include View
		TGridView	TPattern
		TNumberText	
		TTextGridView	
		TTEView	

The rest of this section gives details about creating each type of item.

Button

If the button is not large enough to display the whole title, press Enter to select the button, then reshape it by dragging the reshape box in the lower-right corner with the arrow pointer. (For details, see the later section “Changing or Deleting Items.”)

Check Box

If you want more than one check box, continue to create them by clicking, then typing the title. To align the check boxes, use the Drag to Grid command or the Align Items command in the Options menu.

Radio Button

You can continue to create radio buttons by clicking, then typing a title. To align the radio buttons, use the Drag to Grid command or the Align Items command in the Options menu.

AppMaker groups radio buttons by their item numbers. (You can display item numbers by choosing Show Item Numbers from the Options menu.) If you want radio buttons to be grouped together, create them consecutively with no intervening items. Conversely, if you want multiple groups of radio buttons, create some other item between the groups to separate them, or use the Cut and Paste commands to renumber selected radio buttons after you've created other items in the window. For details on how items are numbered, see the earlier section "Order and Overlapping of Items" in the introductory part of "Working with Items."

Note: To group radio buttons in applications using the THINK Class Library, you must put them inside a labeled group or a radio group pane. If you're using MacApp, you must put radio buttons inside a cluster. For details on creating these, see the heading "Labeled Group" later in this section. A group of radio buttons must be *inside* its enclosure, not behind it (displayed as gray), so make sure you create the radio buttons after the enclosure instead of just drawing an enclosure around existing radio buttons. To correct the order if the radio buttons are behind the enclosure, drag a selection rectangle around the enclosure to select all the items, then Shift-click to deselect the enclosing item, then move the radio buttons. Or drag the enclosure away, select the radio buttons, and drag them inside the enclosure. For details on the front-to-back ordering and the nesting of items, see the earlier section "Order and Overlapping of Items" in the introductory part of "Working with Items."

AppMaker displays the first radio button in each group as "on", the others as "off".

Static Text

As you type

As you type static text, it expands to the right. You can create multi-line text by pressing the Return key to start a new line. The placement of the first Return character determines the width of the item. (Or, if your typing reaches the edge of the Items window, text will automatically wordwrap to the next line, setting the width of the item.) As you continue typing, subsequent text wordwraps according to this current width and expands vertically as needed. You can force a new line by pressing the Return key.

You can reshape the text later and AppMaker will automatically re-wordwrap text to fit in the new width. (For details, see the later section "Changing or Deleting Items.")

Note: The number of characters you can type in static text is limited to 240 characters.

After you press Enter or click elsewhere in the Items window, AppMaker displays a base line for each line of a multi-line item, to help you see its size and to help in aligning with other items. You can turn off the base line display by choosing the Show Base Lines command from the Options menu.

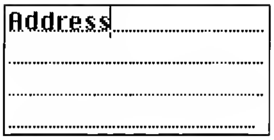
Edit Text

The size that you drag when you create an edit text item is the initial size of the field the user can type in. (You can change the size of the field at any time by reshaping it.) When you release the mouse button, AppMaker displays a rectangle around the edit text field and displays an insertion point for you to type a name for the field. AppMaker uses the name to form the name of a variable in the generated source code (not an initial value for the field). If you supply a meaningful name, the code will be more readable.

If a field is more than one line tall, it is displayed with base lines. You can turn off the base line display by choosing the Show Base Lines command in the Options menu.

For procedural languages, if the field is intended to hold a number at run-time, type a number sign (#) at the end of the field's name. AppMaker will generate code to treat this field as a number rather than as text. (For the THINK Class Library and MacApp, this kind of field is created by viewing the Tools menu as text and using the CIntegerText tool or the TNumberText tool, respectively.)

A multi-line edit text item with base lines displayed



To specify a number field for procedural languages, type a number sign at the end of the field's name

Icon

To create an icon item with the Icon tool, you must have at least one ICON resource already present in your document. The resource can be one that you've created in ResEdit or an equivalent, and placed in the AppMaker document. Or, if you copy an icon (in ICON format) from another application and paste it into an Items window in AppMaker, AppMaker creates both an ICON resource and the icon item for you. The ICON resources in your System file are also available for use in AppMaker. The icon item (created by you with the Icon tool, or by AppMaker when you paste an icon into the Items window) tells the application where to display the ICON resource.

The Icon tool lets you create a **static icon**, which is for visual appearance only. In a procedural dialog, you also have the option of creating an **icon button**, which responds to mouse-down events in the same way as a radio button: the selected icon is highlighted and the other icons in the group are displayed in their normal state.

There is another way to create a static icon besides using the Icon tool: you can copy the icon from the Scrapbook or Clipboard into the Items window. After you choose Paste Icon from the Edit menu, AppMaker pastes the icon in the upper-left corner of the Items window. You can use the Arrow tool to move it to a different location. For details, see the later section "Changing or Deleting Items."

Note: If you're creating icon buttons (available only when using a procedural language), the items must be consecutively numbered, the same as for radio buttons. For details, see the earlier section "Radio Button."

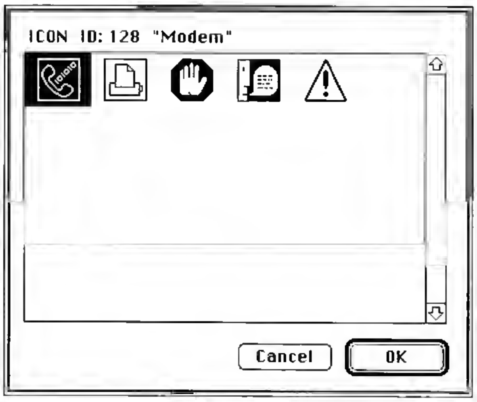
To use the Icon tool, first select it from the Tools menu. Then follow these steps:

1. To create a static icon, click where you want the upper-left corner of the icon.

   OR

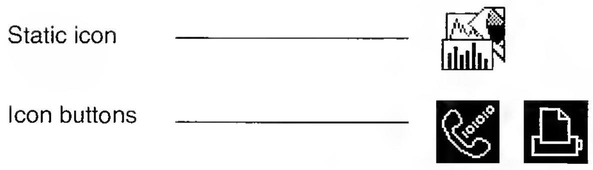
   To create an icon button, hold down the Option key while clicking.

2. After you click to position the icon, AppMaker displays a graphic list of the available icons from the application file or the System file.



3. Click the icon you want, then click OK.

   AppMaker displays the selected icon. If you created an icon button, AppMaker displays it inverted to differentiate it from a static icon.



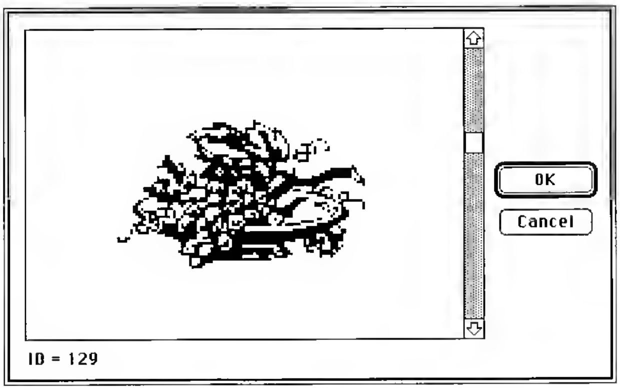
Note: For an alert, you don't have to create an icon item. The System automatically displays the appropriate icon in the standard position in an alert when your code calls one of the library routines, Acknowledge or Confirm, or the Toolbox routines, StopAlert, NoteAlert, or CautionAlert. For details, see the section "Miscellany Routines" in Chapter 5, "Programming a Procedural Application."

Picture

When you use the Picture tool to create a picture item, the picture is static; it can't act as a control. If you want to create a picture that behaves as a control (i.e., responds to mouse clicks), create a custom control instead. For details, see "Custom Control" later in this section.

To create an picture item with the Picture tool, you must have at least one PICT resource already present in your document. The resource can be one that you've created in ResEdit or a graphics application, and placed in the AppMaker document. The picture item tells the application where to display the PICT resource.

After you click with the Picture tool to indicate where you want the upper-left corner of the picture, AppMaker displays a graphic list of the pictures (PICT resources) that are in the AppMaker document. You can scroll the list to select a picture, then click OK to have AppMaker display it in the Items window.



You can also create a static picture in an Items window by copying a picture in PICT format from the Scrapbook or Clipboard (as described earlier in the section "Basic Steps for Creating Items"). After you choose Paste Picture from the Edit menu, AppMaker pastes the picture in the upper-left corner of the Items window, creating a PICT resource in the process. You can use the Arrow tool to move the picture item to a different location. For details, see the later section "Changing or Deleting Items."

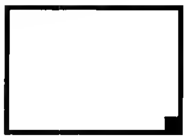
Line

Gray lines are used to visually separate items. With the line tool, you can draw horizontal lines and vertical lines, by far the most common cases. You can also draw diagonal lines, provided that they slope downwards to the right. (You can't draw diagonal lines that slope upwards to the right because the line is defined by the upper-left and lower-right corners of a rectangle.)

Rectangle

Rectangles are used to visually group related items. The items inside a rectangle are nested; when you move the rectangle, items within it move too. Parts of enclosed items that extend beyond the edges of the rectangle aren't shown. For more information on the behavior of nested items, see the earlier section "Order and Overlapping of Items."

To make sure items are within the rectangle, instead of behind it (shown gray), create the rectangle before you create the items it encloses. Or, correct the situation as described in the section "Order and Overlapping of Items."



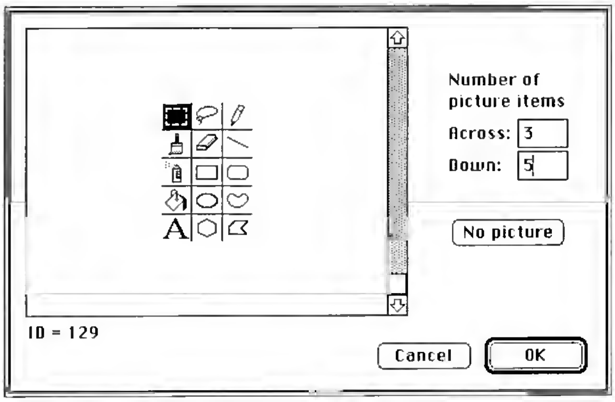
Labeled groups are also used to group items; they are like rectangles with labels. For details, see "Labeled Groups" later in this section.

Palette

A palette is a picture with additional information that describes the "dimensions" of the palette, which must be divisible into equal-sized grid units across and down.

To create a palette with the Palette tool, you must have a PICT resource for the palette already present in your document. As for a picture, you create the visual image of the palette in a graphics application and save it in PICT format. For instructions on making the palette available for use in your AppMaker document, see the earlier section "Basic Steps for Creating Items."

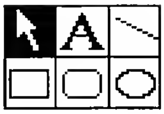
After you click with the Palette tool where you want the upper-left corner of the palette, AppMaker displays a dialog box asking for information about the palette:

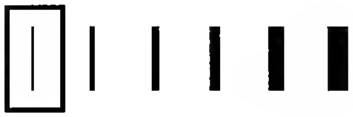


1. Use the scroll bar to display the picture (PICT resource) to use for the palette.
2. In the Across field, type the number of picture items that appear in any row of the palette.
3. Press the Tab key to move the insertion point to the Down field, then type the number of picture items that appear in any column of the palette.

4. Press the Tab key to move the insertion point to the Frame Width field, then type an integer value for Frame Width.

   If you type a zero (0), the selected item in the palette will be displayed inverted. If you type a number greater than zero, a rectangular frame will be drawn around the current selection, using a pen width of the specified number of pixels (individual dots on the screen).


Frame width of zero causes selected item to be displayed inverted


A frame width greater than zero (in this case, 2) determines the line width of the frame drawn around the selected item

5. Click OK to accept the palette definition.

   AppMaker draws the palette. The first item in the palette is highlighted to show its run-time appearance and to emphasize that this is a palette, not just a picture.

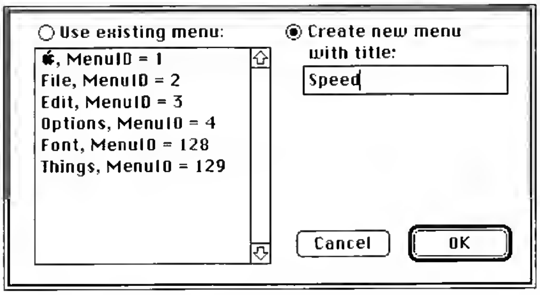
Pop-up

In accordance with the Apple *Human Interface Guidelines*, a pop-up menu should have a label that identifies it. This label is, by convention, immediately to the left of the pop-up menu. When you create a pop-up menu item, AppMaker creates a static text label with the same name as the menu. If you're using a procedural language, you can edit the label, move it, or delete it. If you're using the THINK Class Library or MacApp, the text of the label can be edited by using AppMaker's menu editor to change the name of the corresponding submenu, but the label can't be separated from the pop-up. (If you're using the THINK Class Library you can create a pop-up with no label by viewing the Tools menu as text and using the CArrowPopupPane tool instead of the CStdPopupPane tool.)

When you create a pop-up menu item, you can reference an existing menu, or you can have AppMaker create a new submenu for you. In either case, you edit the items in the menu the same way as you edit any other menu—you use AppMaker's menu editor as described in the earlier section, "Working with Menus."

A pop-up menu can be a picture menu or a text menu. For details on creating a picture menu, see "Working with Menus."

After you've clicked to indicate where you want the upper-left corner of the pop-up, AppMaker displays a dialog box asking you to select an existing menu or create a new menu:



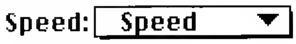
◯ Use existing menu: [✓ ◉ Create new menu with title: Speed]

, MenuID = 1
File, MenuID = 2
Edit, MenuID = 3
Options, MenuID = 4
Font, MenuID = 128
Things, MenuID = 129

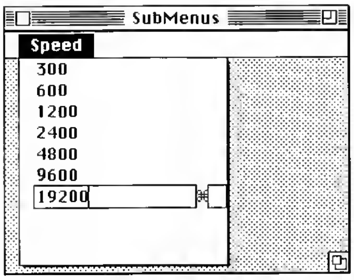
1. If you've already created the menu, select the menu in the left-hand list, then click OK. AppMaker displays a static text label identical to the menu title, and a pop-up box.

   OR

   To create a new menu, type a title for the new submenu at the insertion point, then click OK. AppMaker displays a static text label from the menu title, and a pop-up box.



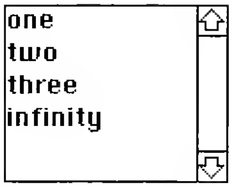
   If a SubMenus menu bar does not already exist, AppMaker creates it. AppMaker then creates the new submenu in the SubMenus menu bar. To add menu items to the newly created menu, see the earlier section, "Working with Menus."



List

With the List tool, you can create the most common form of list: a one-column list with a vertical scroll bar. You can create other kinds of lists by modifying the AppMaker library or the generated code.

The contents of the list at run-time are, of course, defined by your code. For details on where to add your code for the list, see Chapter 5, "Programming a Procedural Application," or Chapter 6, "Programming a THINK Class Library Application."



Note: For languages using the THINK Class Library, the List tool creates just a CArrayPane item; for MacApp it creates just TTextListView. A complete scrolling list requires the creation of three nested items: for TCL, a CAMBorder enclosing a CScrollPane enclosing a CTable or CArrayPane; for MacApp, a TControl enclosing a TScroller enclosing a TTextListView. Instead of using the List tool, follow these steps:

1. For TCL:

   ☐ Use the Rectangle tool to drag a rectangle the size you want the list, starting at the upper-left corner.

   ☐ Use the ScrollPane tool to create a scroll pane inside the rectangle, the same size as the rectangle.

   ☐ With Drag to Grid off (unchecked) in the Options menu, use the List tool to drag a list starting one pixel below the top and left of the scroll pane, to fill the area just inside the scroll bars.

   OR

For MacApp:

- ☐ Use the Rectangle tool to drag a rectangle the size you want the list, starting at the upper-left corner.
- ☐ Turn Drag to Grid off in the Options menu,.to allow precise positioning of the scroller.
- ☐ Use the Scroller tool to drag a scroller inside the rectangle, starting one pixel below the top and left of the rectangle, to fill the area just inside where the vertical scroll bar will be. (MacApp creates its scroll bars outside the scroller.)
- ☐ Use the List tool to drag a list from the upper-left corner of the scroller to the lower-right corner inside the scroll bar.

2. If you were using the grid to position items, turn Drag to Grid back on by choosing Drag to Grid from the Options menu.

Scroll Bar

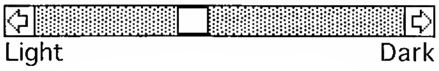
Scroll bars are used to scroll a window vertically or horizontally, as controls for setting a value (such as the Brightness control in Apple's color wheel dialog box for choosing colors), or, for procedural languages, to scroll an item within a window if you add code for the behavior of the item.

Note: If you're using the THINK Class Library or MacApp, you use the Scroller tool instead of the Scroll Bar tool to create a scrollable region in a window. (If you're viewing the Tools menu as text, the THINK Class Library tool is called CScrollPane; the MacApp tool is called TScroller.) You can add a scroll bar to a scroller by clicking with the Scroll Bar tool on the edge where you want the scroll bar. For details on scrollers, see "Scroller/Scroll Pane" later in this section.

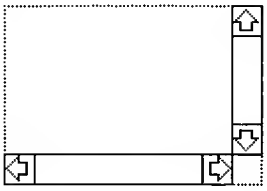
You don't have to be precise when you drag to create a scroll bar. AppMaker always creates it with the standard width or height. If you drag a scroll bar so that it is taller than it is wide, AppMaker will make it a vertical scroll bar with standard width. Similarly, if you make it wider than it is tall, it will be given standard height.

For procedural languages, if you create a scroll bar that touches the right edge of the window, AppMaker will recognize that it is a window's vertical scroll bar and extend the bottom of the scroll bar down to where the size box will be. If you create one touching the bottom edge, AppMaker will extend its right edge to the size box.

A scroll bar can be used as a control

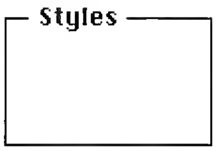


To add a scroll bar to a scroller, click on an edge with the scroll bar tool



Labeled Group

A labeled group is similar to a rectangle, in that it can be used to group elements in an Items window. It is smaller than its analogous rectangle, providing space for a label at the top, which fits within the rectangle.



In the THINK Class Library, a labeled group is used to define radio button groups. (The CRadioGroupPane tool, available when you view the Tools menu as text, can also be used to define TCL radio button groups. It displays neither a rectangle nor a label.) In MacApp, this grouping tool is referred to as a cluster.

To make sure items are *within* a labeled group instead of behind it (displayed as gray), create the labeled group first, then create the items inside it. For details on this or on the behavior of nested items, see the earlier section "Radio Button", or the section in beginning of "Working with Items" called "Order and Overlapping of Items."

For TCL and MacApp, if you don't type a label for a labeled group or cluster, the group itself is invisible, but still acts as a container for the items nested within it.

For Procedural languages, a labeled group is actually implemented as two separate items: a rectangle, and a static text item. If you don't want a label you can delete it, or just use the rectangle tool. You don't need to use the labeled group tool because procedural languages group radio buttons by their item numbers.

Scroller/Scroll Pane

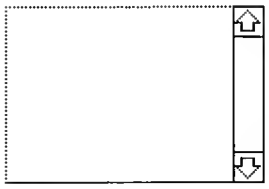
Note: Scrollers are not available for procedural languages. Instead, use the Scroll Bar tool as described earlier in the section "Scroll Bar." (Scroller is the MacApp name; Scroll Pane is the THINK Class Library name. For simplicity, this description uses the name Scroller.)

Scrollers are freestanding scroll regions that can have either no scroll bars (such as the original MacPaint window which was scrolled using the Hand tool), a horizontal scroll bar, a vertical scroll bar, or both. With a scroller, you can create a scroll bar control at the edge of a window (without the scroll bar scrolling the window contents), or have a scroll bar that extends only part way along a window edge, or scroll something within the window that is not single-line text (which is what the List tool creates).

When you drag to define the size and shape of the scroller, AppMaker creates it with a vertical scroll bar. You can add a horizontal scroll bar, if needed, by using the Scroll Bar tool and clicking on the bottom edge where you want the scroll bar. The scroll bars and the scroll region are each selectable, so you can delete a scroll bar without affecting the scroll region. If you're using MacApp, you can also resize the parts independently, and move the scroll bars along the edge of the scroll region by dragging them. (In the TCL, the scroll bars must match the height and width of the scroll region, so the scroller is resized as a unit.)

The scroll region is drawn with a dotted line to distinguish it from a rectangle.

A scroller is created with a vertical scroll bar



Custom Control

A Custom Control item is an item that refers to a separate control defined as a CNTL resource in the application's resource fork. For procedural languages, AppMaker implements scroll bars and palettes as custom controls but this is invisible to you because AppMaker creates the associated CNTL resources.

You can use the Custom Control tool to place a control for which you've created your own CNTL resource (with ResEdit or equivalent), or you can create one of five built-in custom controls for which AppMaker has a custom control definition (CDEF). When you create any of these built-in custom controls, AppMaker creates a control (CNTL) resource for you and copies the CDEF to your application.

The built-in custom controls have behaviors associated with familiar user interface objects; you supply the picture or pictures that customize the control. The five custom controls are:

- Picture button
- Picture check box
- Picture radio button
- Multi-picture button
- Slider

The first three behave just like a standard button, check box, and radio button, but substitute the appearance of those standard controls with an "Off" picture and an "On Picture" that you supply to represent the two states the control can have.

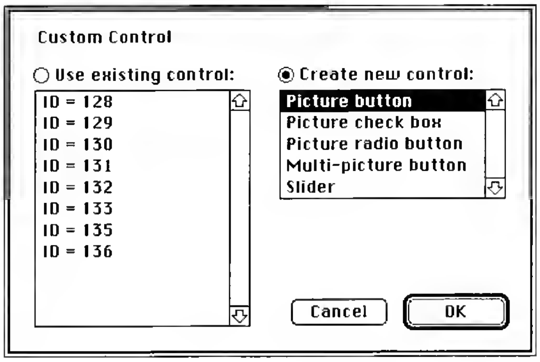
A multi-picture button, instead of allowing just an "Off" state and an "On" state, allows up to 100 states, which are displayed in sequence with each click of the mouse, cycling back to the first state after the last state is clicked. (Your code can change this default behavior, if needed.) You supply a picture for each state you want the control to have.

A slider lets the user drag an indicator to set a value. In the Control Panel (called Sound in System 7), the Speaker Volume control is an example of a slider. In some applications, the user drags a gray outline of the indicator; in others, he or she drags the indicator itself. Sometimes the indicator is constrained so it can be dragged only to discrete points; sometimes the indicator can be dragged to any point but, when the user releases the mouse button, the indicator jumps to a discrete setting. AppMaker lets you create any of these kinds of sliders: you supply a picture for the slider and one for the indicator, and choose the drag behavior.

As with other items or menus that use pictures, you need to create the pictures in PICT format, in the size needed, in a separate graphics application. The size of off/on pictures or states for a multi-picture button should have the same dimensions. The white areas (i.e., the non-colored pixels) in pictures for any picture control must be opaque, not transparent.

To use the pictures in custom controls, you can paste the pictures from the Scrapbook or clipboard into an Items window and then delete them, as described in the earlier section "Basic Steps for Creating Items." Deleting a picture item deletes the item but doesn't delete the PICT resource, which is still available for use in custom controls.

After you've clicked where you want the upper-left corner of the control to appear, AppMaker displays a dialog box asking you to select an existing CNTL resource or create a new custom control.


Custom Control

Use existing control: [ ] Create new control: [✓]

ID = 128
ID = 129
ID = 130
ID = 131
ID = 132
ID = 133
ID = 135
ID = 136

Picture button
Picture check box
Picture radio button
Multi-picture button
Slider

To create a control from an existing CNTL resource, select a control from the left-hand list, then click OK (or double-click the control). That's all there is to creating the control; AppMaker draws it in the Items window.

To create a new custom control of one of the built-in types, double-click on the type of control you want to create. Then complete the dialog box that appears. The steps are described in more detail below for the different types of control:

Picture button, picture check box, or picture radio button:

Picture buttons

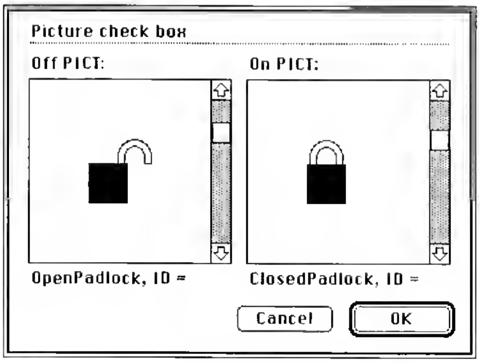
Picture check box

Note: If you're creating picture radio buttons using the THINK Class Library, you must enclose them in a labeled group or a radio group pane, as for regular radio buttons. If you're using MacApp, the picture radio buttons must be enclosed in a cluster. For more information, see the earlier sections "Radio Button" and "Labeled Group."

1. In the list under the radio button labeled "Create New Control", click Picture Button, Picture Check Box, or Picture Radio Button according to what you want to create, then click OK (or double-click the type of control).

2. AppMaker displays a dialog box asking you to select two pictures. Two lists display available pictures (PICT resources) from the application file. The left one is for selecting the "off" picture; the right, the "on" picture.

3. Use the scroll bars to select the Off picture and the On picture, then click OK.



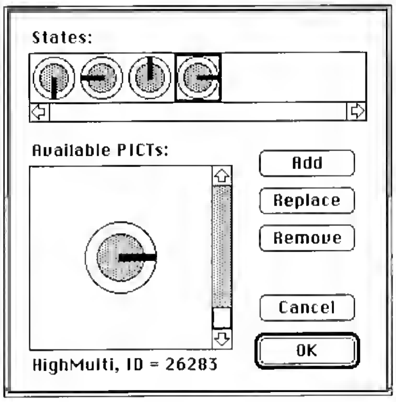
AppMaker displays a picture control in the Items window.

Multi-picture button:

At run-time, each click on a multi-picture button changes its state

1. In the list under the radio button labeled "Create New Control", click Multi-picture Button, then click OK.

2. AppMaker displays a dialog box asking you to select a set of pictures (PICT resources) to represent each state the button can have.



3. Select the pictures for the multi-picture button:

   Note: For the best appearance, all pictures in the set should have the same dimensions. Pictures of different sizes will be scaled to the dimensions of the first picture in the set.

   - ☐ To begin creating the set of pictures, use the scroll bar in the Available PICTs list to display the picture to add, then click Add.

     The picture is displayed in the sequential list of states across the top of the dialog.

- ☐ To add a picture in the middle of the set, select the picture before the one you want to add in the list of states across the top. Then use the scroll bar in the Available PICTs list to display the picture to add, and click Add.

  The picture is added after the selected item in the list across the top.

- ☐ To replace a picture in the set, select it from the list of states across the top, and select the replacement picture from the list of available pictures. Click Replace.

  The replacement picture is displayed in the list of states.

- ☐ To remove a picture from the set, select the picture from the list across the top of the dialog, then click Remove.

  The picture is removed from the list.

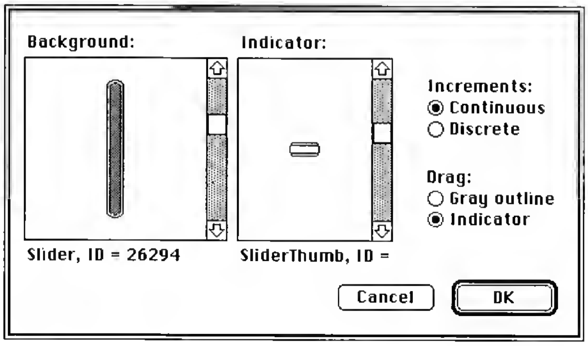
4. When the set of pictures is complete, click OK.

   AppMaker displays a multi-picture button in the Items window, using the first state picture.

Custom slider:

1. In the list under the radio button labeled "Create New Control", click Slider, then click OK.

2. AppMaker displays a dialog box asking you to select a background and an indicator from the available PICT resources.



Background:
Slider, ID = 26294

Indicator:
SliderThumb, ID =

Increments: [✓ Continuous] Discrete

Drag: Gray outline [✓ Indicator]

3. Use the scroll bars to select a picture for the background and a picture for the indicator.

4. Use the top set of radio buttons to choose whether increments are continuous or discrete:

   To let the indicator be dragged to any pixel, click the Continuous button; to constrain dragging to discrete intervals, click the Discrete button.

5. Use the lower set of radio buttons to choose whether to drag a gray outline of the indicator or to drag the indicator itself.

6. Click OK.

   AppMaker displays the slider in the Items window.

User Item

?

A user item is a hook to allow the programmer to do anything that can't be done with the normal tools. In procedural languages, AppMaker implements gray lines, rectangles, pop-up menus, and scrollable lists as user items but this is invisible to you because AppMaker generates all of the necessary code to implement those "user items." (In the THINK Class Library and MacApp, these items are built in classes.) If you create a user item with the User Item tool, you'll need to supply the necessary code as documented in *Inside Macintosh*, Chapter 13, "Dialog Manager." When you release the mouse button after dragging with the User Item tool, AppMaker draws a rectangle with a thick gray border to indicate the user item.

Changing or Deleting Items

After creating an item, you change it in any of the following ways:

- ☐ Move the item
- ☐ Align it with other items
- ☐ Reshape the item
- ☐ Edit the title
- ☐ Change the text style of any text associated with the item
- ☐ Display and change information such as the enabled or disabled status of the item, its coordinates and size, and, if you are using the THINK Class Library or MacApp, its class name. (For MacApp, Item Info also lets you see and change the ViewID and whether the item is a target.)
- ☐ Delete the item

To do any of these operations, you first use the Arrow tool to select one or more items. Reshaping, editing a title, and getting Item Info operate on a single selected item. Moving, changing text style, and deleting operate on one or more selected items. Aligning operates on two or more selected items.

The selection rules are the same as the Finder's rules for selecting icons on the desktop. Selected items are outlined and have a black reshape box in the lower-right corner.

To select an item:

1. Using the Arrow tool, click the item.

   Any previously selected items are deselected and the selected item is highlighted.

   Note: Selecting a particular item in a group of overlapping items can sometimes be difficult. A useful technique is to drag a selection rectangle to select multiple items, then hold down the Shift key to click unwanted items until only the item you want is selected. (If you intended the item to be nested, instead of overlapping, moving the item while it is selected should correct the situation. See also the note below.)

To select a range of items:

1. Position the pointer outside the items you want to select.
2. Using the Arrow tool, drag a selection rectangle around the items you want.

   When you press down on the mouse button, any previously selected items are deselected.

   When you release the mouse button, any items which have any part inside the selection rectangle are selected.

   Note: Nested items are considered to be in front of their parent, so they receive mouse clicks first. If items are behind, instead of within, the parent item, you can correct the situation by dragging a selection rectangle around the parent item to select all the items, then Shift-clicking to deselect the parent, then moving the items to be nested. For details on the front-to-back ordering and the nesting of items, see the earlier section "Order and Overlapping of Items" in the introductory part of "Working with Items."

To add to or subtract from a selection:

1. Hold down the Shift key.
2. Use either of the above selection methods, either clicking or dragging a selection rectangle around additional items.

   If any of the newly selected items were already selected then they are deselected; otherwise they are added to the previous selection.

To move items:

If Drag to Grid is checked in the Options menu, the selected items will be moved to the nearest grid intersection (a multiple of 4 pixels from the upper-left corner of the window).

Note: Nested items are moved with their parent. For example, items inside a labeled group are also moved when you move the labeled group item. If you don't want the nested items to be moved, cut them from inside the parent item, move the parent item, then paste the other items. You could also drag the nested items outside the parent, then move the parent item, and move the other items back into place.

1. Using the Arrow tool, select one or more items.

2. To see the coordinates of the item as you move it, choose Item Info from the View menu and leave the Item Info window open as you move the item.

3. With the Arrow on any selected item, drag the item(s) where you want them. As you drag, a gray outline of the item(s) follows the pointer. When you release the mouse button, the item(s) are drawn in their new position.

   OR

   Press a keyboard arrow key to move the item(s) one unit in the appropriate direction.

   If Drag to Grid is checked in the Options menu, each keystroke will move the item(s) four pixels. If it is unchecked, each keystroke will move them one pixel.

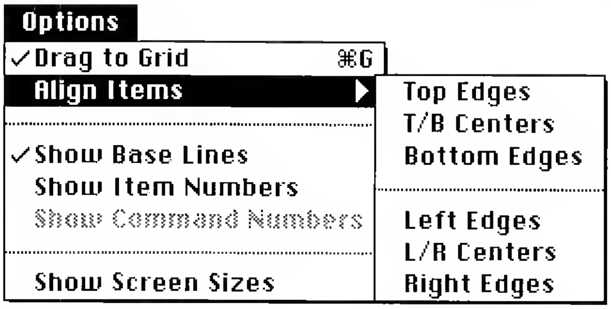
To constrain movement of items to be horizontal or vertical:

1. Hold down the Option key as you start to drag the items.

   The items will move only horizontally or only vertically depending on the direction in which you first move the mouse.

To align items:

You can align items either by their edges (top, bottom, left, or right), or by their centers (top/bottom centers or left/right centers).



Items are aligned to the lowest-numbered item of the selected items. (For details on item numbering, see the earlier section "Order and Overlapping of Items.")

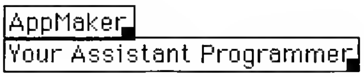
If Drag to Grid is checked in the Options menu, the selected items will be moved to the nearest grid intersection (a multiple of 4 pixels from the upper-left corner of the window).

1. With the Arrow tool, select the items you want to align.

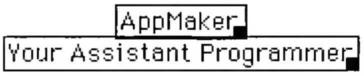
2. From the Align Items hierarchical menu in the Options menu, choose which edges or centers you want to line up.

   AppMaker aligns the items according to what you've chosen.

Aligned by Left Edges



Aligned by L/R Centers



To reshape an item:

If Drag to Grid is checked in the Options menu, the selected item will be sized to a multiple of four pixels in both height and width. To reshape an item to any size, ignoring grid boundaries, make sure Drag to Grid is unchecked before you reshape an item.

Note: If you're using the THINK Class Library or MacApp, AppMaker supports the auto-sizing attribute settings. If you reshape an item so that its right or bottom edge touches the right or bottom edges of its parent, AppMaker sets and interprets the attribute settings to auto-resize the item to the height and/or width of its parent (depending on whether the item touches just the bottom edge, just the right edge, or both edges) when the parent is resized. For example, you can resize a list by selecting all of its parts and dragging the size box of the parent item; the nested items are automatically resized accordingly.

1. Using the Arrow tool, select an item.

   The selected item will have a reshape box in its lower-right corner.

2. To see the size of the item (in pixels) as you reshape it, choose Item Info from the View menu and leave the Item Info window open as you reshape the item.

3. Drag the reshape box until the item is the size you want.

   As you drag, a gray outline of the item follows the pointer.

   When you release the mouse button, the item is drawn with its new size. If the item is an icon or picture, it is scaled to fit the new size.

To edit the title of an item:

1. Using the Arrow tool, select the item.
2. Click in the title where you want an insertion point.

   The arrow changes to an insertion point where you clicked. The reshape box disappears until you're finished editing.

3. Edit the title. You can use all of the standard text editing methods including Cut, Copy, Paste and Undo.
4. Press the Enter key to finish changing the title.

To change text style:

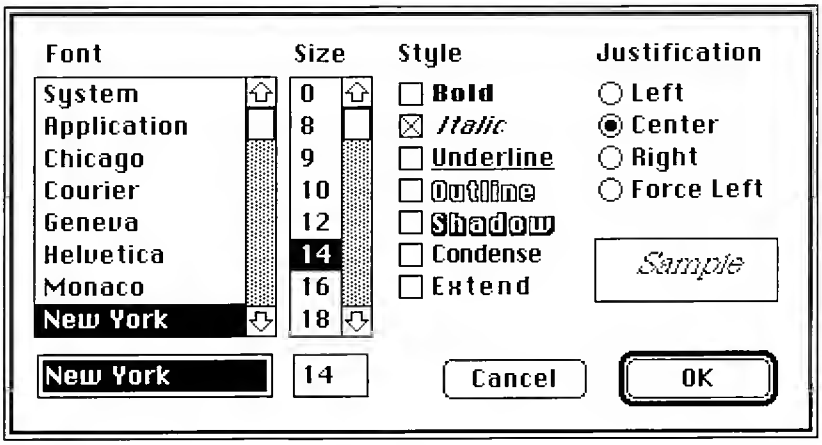
You can change the text style of one or more selected items. Text style includes Font, Size. Style, and Justification.

Note: Not all settings are available for all items. For example, in procedural languages, text style applies to static text and edit text, not to controls. In the THINK Class Library and MacApp, the Style and Justification settings may not apply to all items.

1. Select the item(s) for which you want to change the text style.

2. Choose Text Style from the Edit menu.

   AppMaker displays the Text Style dialog box.



Font: System, Application, Chicago, Courier, Geneva, Helvetica, Monaco, [✓ New York]
Font field: New York

Size: 0, 8, 9, 10, 12, [✓ 14], 16, 18
Size field: 14

Style: ☐ Bold, ☒ Italic, ☐ Underline, ☐ Outline, ☐ Shadow, ☐ Condense, ☐ Extend

Justification: ○ Left, [✓ Center], ○ Right, ○ Force Left

3. Change the Font, Size (in points), Style, or Justification as needed.

   - ☐ Font: You can select the preset System or Application font or a font named in the list, or type the name of a font that is not currently installed but will be available on the user's system. (For Roman-language systems, the System font is Chicago, and the Application font is Geneva.)

   - ☐ Size: You can select a size from the list, or type an integer for the point size you want. A size of zero (0) refers to the standard size for the font. For example, for U.S. systems, the standard System font size is 12 point.

   - ☐ Style: Click one or more check boxes for the style(s) you want.

   - ☐ Justification: Click a button for the setting you want. (The Force Left choice for Justification causes an item to be left-justified even when the interface is translated to a language that reads from right-to-left, whereas the choice of Left would cause the item to change to right-justified in a right-reading language.)

   AppMaker displays a sample of the text style you've chosen in a box above the OK button.

4. When you're satisfied with the text style choices, click OK.

   AppMaker displays the selected item(s) using the new text style.

To display or change item information using Item Info:

In addition to the graphic display of items in a window, you can display information about a selected item in numeric form. This lets you position or reshape an item to precise coordinates or size, or view the numeric information as you move or reshape an item in the Items window. The Item Info dialog box also lets you display and change the enabled/disabled status of an item, and, for the THINK Class Library and MacApp, the class name. For MacApp, Item Info also lets you see and change the ViewID and whether the item is a target (i.e., takes control when the window is frontmost; this is usually an Edit Text item. If you don't specify a target for a dialog view, AppMaker sets the TDialogView as the target.).

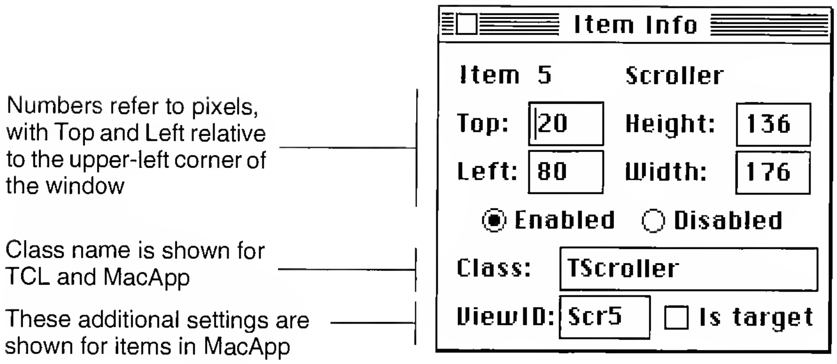
Note: If you want to see how moving or reshaping an item looks in relation to the standard screen sizes of the Macintosh Classic and Macintosh II, use the Show Screen Sizes command in the Options menu.

1. Using the Arrow tool, select the item.

   Note: To see Item Info, only one item can be selected. In the THINK Class Library and MacApp, some items are actually made up of several parts. For example, a scroll pane is created with a scroll pane and a horizontal scroll bar. Both of these are selected at the time of creation. One at a time should be selected to see Item Info.

2. Choose Item Info from the View menu.

AppMaker displays the Item Info dialog:


Numbers refer to pixels, with Top and Left relative to the upper-left corner of the window
Class name is shown for TCL and MacApp
These additional settings are shown for items in MacApp
Item Info

Item 5 Scroller

Top: 20 Height: 136

Left: 80 Width: 176

(● Enabled) ○ Disabled

Class: TScroller

ViewID: Scr5 ☐ Is target

3. Change Item Info as needed:

   □ To move the item, type new coordinates into the Top and/or Left fields.

   AppMaker moves the item to the new position.

   □ To reshape the item, type a new Height and/or Width.

   AppMaker draws the item with its new shape.

   □ To change the enabled/disabled status of an item, click the Enabled or Disabled button.

   AppMaker redraws the item. Disabled buttons, check boxes, or radio buttons are drawn dimmed. Enabled icons are drawn inverted.

   □ If you're using the THINK Class Library or MacApp, you can change the item's class name by typing a new name in the Class field.

   □ If you're using MacApp, you can also change the item's ViewID by typing a new ID in the ViewID field, and specify whether the item is a target by clicking the check box labeled Is Target.

4. To display info for a different item, click in the Items window to select it, then, using the Arrow tool, select the item.

   AppMaker updates the Item Info dialog to display information for the newly selected item.

To delete items:

Note: Deleting a labeled group or radio group pane in the THINK Class Library, or a cluster in MacApp, deletes the grouping mechanism for radio buttons and consequently the application's ability to recognize any radio buttons the group contains. If you just want a labeled group or cluster to be invisible (no visible border), but retain the grouping mechanism, delete the label for the group by using the Arrow tool to click in the label, then selecting and deleting all the characters in the label.

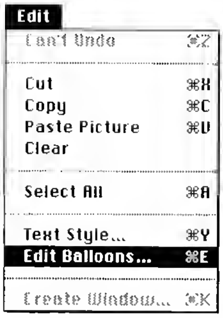
1. Using the Arrow tool, select one or more items.

2. Choose Clear from the Edit menu.

   Or, press the Delete key, or choose Cut from the Edit menu.

   The selected items are deleted, the Items window is redrawn, and the remaining items are renumbered.

Editing Balloon Help



After you've designed the user interface to your application, you'll probably want to add Balloon Help for many interface elements, so users can browse through your application and find out the intended purpose of the menu commands, icons, and other elements.

AppMaker provides a quick, straightforward way to add the standard Balloon Help (the System 7 feature that the user accesses by choosing Show Balloons from the Help menu, under the question-mark icon at the right end of the menu bar).

By using the Edit Balloons command in AppMaker's Edit menu, you can write help balloons for the following elements:

- Menu titles
- Menu items
- Any item in an Items window except for MacApp subviews

Displaying and Writing the Contents of Balloons

Balloon Help for any given interface element is designed to handle up to four states:

- Enabled
- Disabled
- Checked
- Marked (meaning marked with something other than a check mark)

AppMaker displays all four states at once so you can see which ones you've already written help for, and so you can easily copy and paste from one balloon to another information that is relevant for more than one state.

Not all four states apply to all elements. For example, the Marked state would never be relevant for static text. Usually it will be obvious to you what states apply to a given element.

Note: Items in procedural windows and in THINK Class Library windows or dialogs make use of only one state which is the Enabled state.

Technical Note: A control that is not in a disabled state will display the Balloon Help for either the Enabled, Checked, or Marked states according to the *value* of the control. If the value is 0 (zero), it will display the Enabled help; if the value is 1, it will display the Checked help; and if the value is anything other than 0 or 1, it will display the Marked help. For example, unless you change the values, Checked is used for the first picture in a palette, the lowest setting of a scroll bar control, and the first state of a multi-picture button.

For details on Balloon Help, refer to *Inside Macintosh*, Volume VI, the chapter on the Help Manager.

To display or write Balloon Help:

Note: Because Edit Balloons is a modeless dialog, changes you make take effect immediately rather than when you press an OK button. Therefore, you might want to make a backup copy of your document before making extensive changes to Balloon Help text.

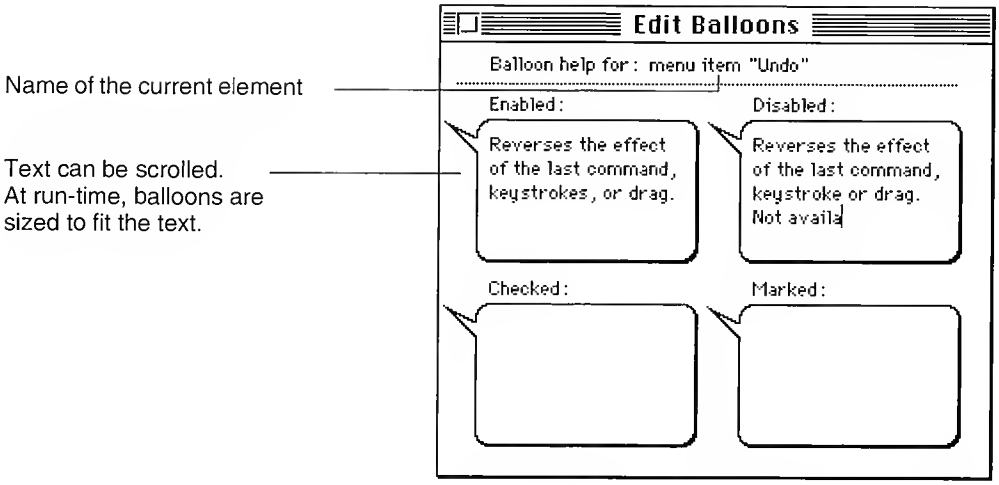
1. Select the element you want to work with by first choosing the kind of resource from the Select menu, and clicking a resource in the List window. Then select a single element such as a menu title or an item.

   For details on selecting a particular interface element, refer to the earlier sections "Working with Menus," "Working with Windows, Dialogs, and Alerts," or "Working with Items."

   Note: To specify Balloon Help for pictures in a picture menu, display the menu as text. The help you specify for the text commands will be displayed when the menu is used as a picture menu.

2. Choose Edit Balloons from the Edit menu.

   AppMaker displays a modeless dialog box containing four representations of balloons, one for each potential state of an interface element. The name of the current element appears at the top of the dialog box.


Name of the current element
Text can be scrolled. At run-time, balloons are sized to fit the text.
Edit Balloons
Balloon help for : menu item "Undo"
Enabled :
Reverses the effect of the last command, keystrokes, or drag.
Disabled :
Reverses the effect of the last command, keystroke or drag. Not availa
Checked :
Marked :

3. To write Balloon Help, type text at the insertion point, or select text to replace. You can use all the standard text editing techniques, including most of the commands on the Edit menu.

   You can type up to 255 characters for each balloon. At run-time, the size of the balloon is adjusted to the amount of text within it.

4. You can continue to select different elements and display or write Balloon Help for them as needed. AppMaker updates the dialog box to reflect the name and any Balloon Help for the current element.

5. When you are finished with Balloon Help editing, choose Close from the File menu, or click the Close box in the Edit Balloons dialog box.

To change or delete Balloon Help:

At any time, you can change or delete the Balloon Help text you have written for any or all of the states of an interface element. Just display the Balloon Help for the element as described above and edit the text with the standard text-editing techniques, or clear the text by positioning the insertion point in a balloon, choosing Select All from the Edit menu, then choosing Cut or Clear from the Edit menu.

Generating Source Code for the User Interface

AppMaker's user interface editor simulates the appearance of all the elements of a user interface as closely as possible to take the guess-work out of designing. However, to test every aspect of the user interface, such as pulling down pop-up menus, clicking check boxes on and off, and so on, you need to have AppMaker generate source code for the user interface, then compile and link it using your language system. Then, when you open your application, you can see exactly how the user interface will look and behave.

You can have AppMaker generate source code at any point during the design process, for either the whole application or for selected modules.

Changing the Programming Language

Choosing a programming language based on your language system is one of the steps in creating an AppMaker document. (For information about this, refer back to the earlier sections "Creating and Opening Applications" and "Choosing a Language.") This step is needed in order for AppMaker to copy certain standard resources appropriate for your selected language into the new application.

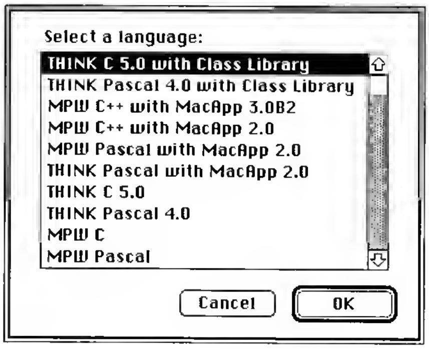
The resources required by each category of language (procedural, object-oriented using the THINK Class Library, and object-oriented using MacApp) are different, so any language change you make after creating your application should be within the same language category.

For example, you can change from THINK C with Class Library to THINK Pascal with Class Library, or from MPW Pascal to THINK Pascal or to THINK C without any problems. If, however, you switch from one category to another, e.g., from MPW C to THINK C with Class Library, your generated program may not work because needed resources are missing.

To change the language:

1. Choose Language from the File menu.

   AppMaker displays a dialog box asking you to select a language system. The language you chose when you created the file is highlighted in the list.


Select a language:
THINK C 5.0 with Class Library
THINK Pascal 4.0 with Class Library
MPW C++ with MacApp 3.0B2
MPW C++ with MacApp 2.0
MPW Pascal with MacApp 2.0
THINK Pascal with MacApp 2.0
THINK C 5.0
THINK Pascal 4.0
MPW C
MPW Pascal
Cancel
OK

2. Click the language you want, then click OK.

   If you try to change from one language category to another, AppMaker displays an alert informing you that the selected language needs different resources. If you go forward with the change (by clicking OK in the alert), you will need to manually copy the appropriate resources or modify the library routines and/or generated code.

   AppMaker stores your selection in the application document. The language you selected will be used for this application until you select a different language.

Generating Source Code

AppMaker generates a separate source code module for each window, dialog, or menu, and for certain other functions as described later in Chapter 5, “Programming a Procedural Application,” and Chapter 6, “Programming a THINK Class Library Application.” You can select which modules to generate. When creating a new application, you'll normally generate all of the modules; when enhancing an existing application, you'll probably generate only those for the enhancements you just created, such as new dialogs or menus.

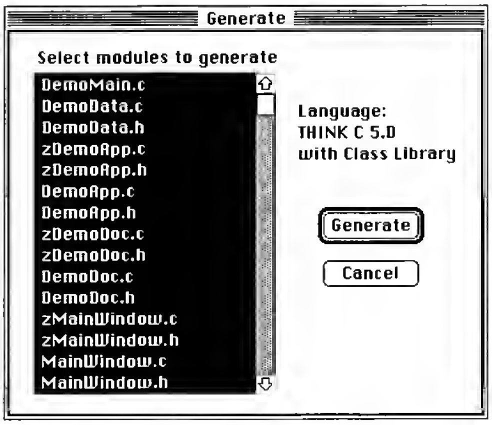
To generate code:

Note: AppMaker generates the source code modules in the same folder as the AppMaker document, so you may want to move the AppMaker document to an appropriate location before starting to generate code.

1. In the AppMaker document, choose Generate from the File menu.

   AppMaker displays the Generate dialog box.

   Note: If you have not already selected a language, AppMaker first displays the Language dialog box so you can select a language.



DemoMain.c
DemoData.c
DemoData.h
zDemoApp.c
zDemoApp.h
DemoApp.c
DemoApp.h
zDemoDoc.c
zDemoDoc.h
DemoDoc.c
DemoDoc.h
zMainWindow.c
zMainWindow.h
MainWindow.c
MainWindow.h

Language:
THINK C 5.0
with Class Library

2. Select the modules you want to generate. You can use the same methods you would use to select files in the Finder:

   - To select a single module, click it.
   - To select a range of modules, drag from the first one to the last one.
   - To add to or subtract from the selection, hold down the Shift key or the Command key when you click or drag.

   AppMaker highlights the selected modules.

3. Click Generate.

   The Cancel button changes to read Stop. AppMaker generates the selected modules in the current folder or disk directory. After each module is generated, it is deselected.

   ⌘.

   To cancel generating at any time, click Stop. Or, hold down Command-period (.).

   Note: If any module already exists, AppMaker displays a standard PutFile dialog box so you can supply a different filename, or change the directory. Or, to overwrite the module, click Save, then click Yes to confirm that you want to replace the existing module.

   After all selected modules have been generated, AppMaker dismisses the Generate dialog.

Compiling and Linking

After generating source code, you'll normally want to compile and link it. You'll exit AppMaker, then use your preferred language system to compile the generated sources, to link them with the AppMaker library routines and with the standard system libraries, and to run the resulting application.

The instructions below tell first how to quit AppMaker, then how to use the various language systems to compile, link, and run your application. For convenience, the instructions use the name `YourApp` to refer to your application, and of course, you'll substitute your own name for the application in carrying them out.

To quit AppMaker:

1. Choose Quit from the File menu.

   AppMaker displays a standard Save dialog box.

2. Click Save or press the Enter key to save your work.

   AppMaker returns to the Finder.

To use THINK C:

These instructions cover both THINK C with Class Library and THINK C without Class Library, because the majority of the steps are the same. In each case you'll start with a project that is shipped with AppMaker, add the newly-generated modules, then compile, link, and run. Differences are noted.

Note: If you're not using the Class Library and you did not compile the AppMaker Library when you installed it, do so now. For details, see "Installing the AppMaker Library" at the beginning of this manual.

1. In the Finder, rename `YourApp` to `YourApp`.π.rsrc.

   WARNING: THINK C requires that a project's resource file have the same name as the project with .rsrc appended. If you don't name the resource file with the proper name, your program will crash as soon as you try to run it.

NewProject.π

2. If you're using THINK C without the Class Library, select NewProject.π in the AMLibraryC folder that you copied onto your hard disk during installation. This project already references the standard THINK libraries and the AppMaker library routines.

   OR

Starter.π

   If you're using THINK C with the Class Library, select Starter.π in the AMClassLibC folder that you copied onto your hard disk during installation. This project already references the standard THINK classes and the AppMaker classes.

3. Choose Duplicate from the File menu.

4. Move the duplicated file (called NewProject.π Copy if you're not using the Class Library, or Starter.π Copy if you are) to the folder containing the AppMaker document and rename it to `YourApp`.π using the Finder.

5. Double-click `YourApp.π` to run THINK C and open up the application's project file.

6. From the Source menu, choose Add.

   THINK C displays a list of the generated files in the current folder.

7. Click Add All, then click Done.

   Note: If you're using THINK C with Class Library, the generated files are added to the main segment. A special file, PlaceHolder.c, is initially the only file in the main segment of the Starter project; it is there only to hold a place for added files. Once you have added other files to the project, you can remove PlaceHolder.c if you want.

8. From the Project menu, choose Run.

   Note: If you want your files to be compiled with debugging code, you can choose Use Debugger before you choose Run.

To use THINK Pascal:

These instructions cover both THINK Pascal with Class Library and THINK Pascal without Class Library, because the majority of the steps are the same. In each case you'll start with a project that is supplied with AppMaker, add the newly-generated modules, then compile, link, and run. Differences are noted.

NewProject.π

1. If you're using THINK Pascal without the Class Library, select NewProject.π in the AMLibraryP folder that you copied onto your hard disk during installation. This project already references the standard THINK libraries and the AppMaker library routines.

   OR

   Starter.π

   If you're using THINK Pascal with the Class Library, select Starter.π in the AMClassLibP folder that you copied onto your hard disk during installation. This project already references the standard THINK classes and the AppMaker classes.

2. Choose Duplicate from the File menu.

3. Move the duplicated file (called NewProject.π Copy if you're not using the Class Library, or Starter.π Copy if you are) to the folder that contains your resource file, project file, and generated code, and rename it to `YourApp`.π using the Finder.

4. Double-click `YourApp`.π to run THINK Pascal and open up the application's project file.

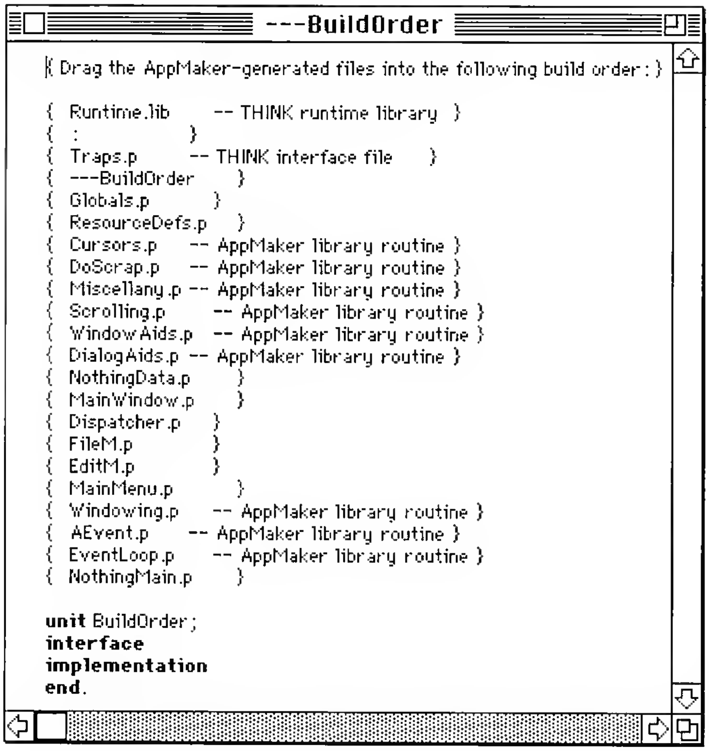
5. From the Project menu, choose Add File.

   THINK Pascal displays a list of the generated files in the current folder.

6. Click Add repeatedly until all of the files have been added, then click Cancel.

   THINK Pascal requires that source files be in the proper order within the project. AppMaker generated a file called "---BuildOrder.p" which lists the files in the proper order.

7. Double-click the BuildOrder.p file to open it.



```
{ Drag the AppMaker-generated files into the following build order : }

{  Runtime.lib      -- THINK runtime library  }
{  :             }
{  Traps.p       -- THINK interface file     }
{  ---BuildOrder     }
{  Globals.p         }
{  ResourceDefs.p    }
{  Cursors.p     -- AppMaker library routine }
{  DoScrap.p     -- AppMaker library routine }
{  Miscellany.p -- AppMaker library routine }
{  Scrolling.p      -- AppMaker library routine }
{  WindowAids.p  -- AppMaker library routine }
{  DialogAids.p -- AppMaker library routine }
{  NothingData.p     }
{  MainWindow.p      }
{  Dispatcher.p    }
{  FileM.p         }
{  EditM.p         }
{  MainMenu.p        }
{  Windowing.p    -- AppMaker library routine }
{  AEvent.p     -- AppMaker library routine }
{  EventLoop.p    -- AppMaker library routine }
{  NothingMain.p     }

unit BuildOrder;
interface
implementation
end.
```

8. Using the BuildOrder.p file as a guide, drag the project files into the proper order.

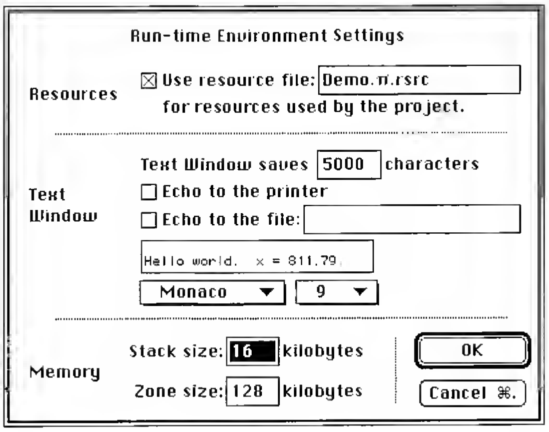
9. From the Run menu, choose Build.

   THINK compiles all of the files.

   Before running your application, you must tell THINK where to find all of the resources (the menus, windows, dialogs, etc.) you created in AppMaker.

10. From the Run menu, choose Run Options.

11. Click the Use Resource File check box, and choose `YourApp.π.rsrc` in the subsequent dialog.



Run-time Environment Settings

Resources
[✓] Use resource file: Demo.π.rsrc
for resources used by the project.

Text Window
Text Window saves 5000 characters
☐ Echo to the printer
☐ Echo to the file:

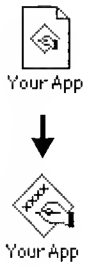
Hello world. x = 811.79

Memory
Stack size: 16 kilobytes
Zone size: 128 kilobytes

WARNING: If you don't designate a resource file for THINK to use, your program will crash as soon as you try to run it.

12. From the Run menu, choose Go.

To use MPW C or MPW Pascal:

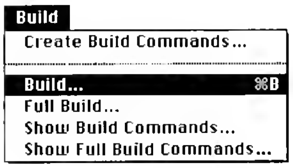


In MPW Pascal or C, the process of linking transforms the original AppMaker document into an application. The linker adds the appropriate code resources to the user interface resources you defined in your AppMaker document.

1. In the Finder, double-click the generated file, `YourApp.make`, to run MPW and to select the containing folder as the current directory.

   Note: The generated source files do not require any editing to compile, link, and run correctly. Opening the file `YourApp.make` is simply a convenient way to run MPW. You might, of course, want to edit a file to add application-specific code.

2. Close the file `YourApp.make`.



3. From the Build menu, choose Build.

4. Type `YourApp`, then click OK.

   If you installed the UserStartup•AppMaker file during installation of AppMaker and if you double-clicked on the file, `YourApp.make`, MPW will already have typed `YourApp` for you.

   MPW's Build program will execute the application's .make file (generated by AppMaker) to compile and link your application. When it is finished, the last line will contain a command to run your program.

5. Press the Enter key to run your program.

To use MPW with MacApp:

1. In the Finder, double-click the generated file, `YourApp.MAMake`, to run MPW and to select the containing folder as the current directory.

   Note: The generated source files do not require any editing to compile, link, and run correctly. Opening the file `YourApp.MAMake` is simply a convenient way to run MPW. You might, of course, want to edit a file to add application-specific code.

2. Close the file `YourApp`.MAMake.

3. In MPW's Worksheet, type `MABuild YourApp`, then press the Enter key.

   You'll see MPW compiling, then linking your `YourApp` program. When it is finished, it will show a line you can execute to run `YourApp`.

Chapter 4
An Example AppMaker Application

This chapter describes an example real-world application, whose user interface was built using AppMaker. We completed this application by adding code to implement its unique features. Though unique, this application contains many features that are common to most Macintosh applications.

Each of the menus, windows, and other user interface elements is described, with specific information concerning how the element should operate, and how it contributes to the overall functionality of the application.

This chapter also covers which of the application's functions were automatically implemented by AppMaker's generated code, and which we needed to implement by adding new code.

Following this discussion, you should have a good feel for what the application is intended to accomplish, and what we did to complete its functionality.

Because AppMaker can generate code for several different target languages, and because the generated code is different for each language, this chapter is followed by additional chapters that discuss the structure and content of the AppMaker generated code for a particular target language. Each discussion focuses on the structure of the generated code and how to locate where individual interface elements are handled in the code. It will also indicate where additional code needs to be added, and how AppMaker's library routines are used in the generated code. We have made use of these routines in our completion of the example application, and you may also use these library routines for the same purpose.

Chapter 5 describes the AMReminder application as implemented in procedural THINK Pascal. The material in that chapter will show the structure of the AMReminder application as generated by AppMaker, and will show the code that we have added to implement the functionality of the AMReminder application in that context.

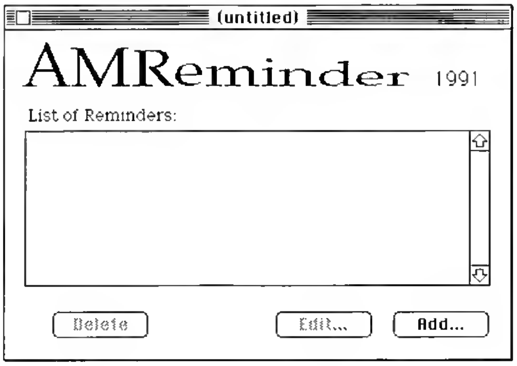
Chapter 6 describes the AMReminder application as implemented in Object THINK C, using the THINK Class Library. The material in that chapter will show the distinct structure of an object-oriented program (OOP) design, as generated by AppMaker. This is quite different from a procedural application.

The Example Application

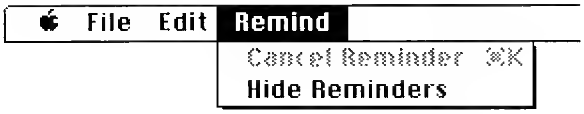
The example we have chosen is a simplified personal reminder application. The application is intended to execute under MultiFinder and provide the following features:

- The ability to handle multiple reminders, each with its own notification features
- Saving and loading reminders to and from disk files
- Flexibility for adding, editing and deleting reminder entries
- Support for various types of notifications, using the features of the Macintosh's Notification Manager
- The ability to hide the main reminder window, to reduce screen clutter
- Background operation that periodically checks for reminders that are due to be posted and interacts with the notification manager.

When the application is launched, the main window appears. It has a single-column list item and three buttons labeled Add, Edit, and Delete.



AMReminder has standard Apple, File, and Edit menus. AppMaker generates code to implement most of the items in these standard menus. AMReminder has one additional menu. The figure below shows the two items on the Remind menu.



We have disabled the first item on the Remind menu, Cancel Reminder, which will be enabled only when a notification is in progress. The second item, Hide Reminders, provides the ability to hide the main window, to unclutter the screen without closing the file. We have written code to change this item to read Show Reminders after the window has been hidden.

We also have written the code to hide and show the main reminder window, when requested to do so. In general, this is simply done by calling an appropriate toolbox routine.

The Edit and Delete buttons in the main window are active or inactive depending on whether a reminder is selected. We have written code to enable or disable them. AppMaker library routines helped in determining whether an item was selected and in enabling the buttons.

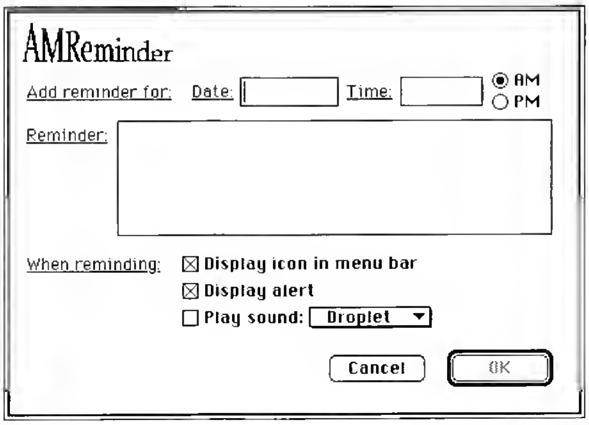
In addition to the buttons, the main window contains a list pane, whose contents is initially empty and whose scroll bar is inactive. AppMaker will generate the code to completely manage scrolling of entries, but we must write the code to format and place entries into the list.

Formatting the entries consists merely of constructing a single string that contains the date, time, am/pm indicator, and the text of the message from the entry dialog's variables. The method for placing the entry into the list differs from language to language; however, in each case, once called, a pre-coded routine (which is either provided as an AppMaker library function, or a method in the THINK or MacApp class library) performs this task.

For example, in the procedural Pascal version of the AMReminder application, we called the Appmaker library routine AddToList to add the abbreviated reminder entry to the scrolling list.

Adding Reminders

New reminders are created by clicking the Add button (which is always enabled) at the bottom of the main window. When this button is clicked, the dialog shown below is presented.


AMReminder

Add reminder for: Date: Time: [✓ AM] PM

Reminder:

When reminding: [✓ Display icon in menu bar]
[✓ Display alert]
Play sound: Droplet

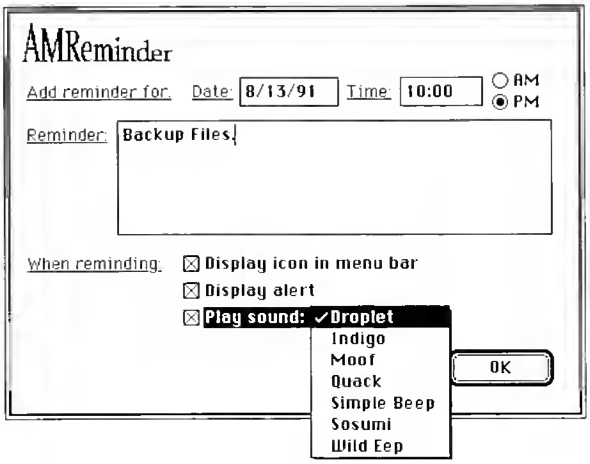
This dialog has text entry fields for a date, time, and message. In addition, it includes check boxes that customize the behavior of the notification to which the entry corresponds. The user can also select a sound to be played, using the pop-up menu labeled Sound. Any sound resource that is presently contained in the System File, or in the resource fork of the AMReminder application will appear in the menu.

We have added code to initialize the state of the check boxes.

The Display Alert check box controls whether AMReminder will use the Notification Manager to display an Alert containing the notification text when a reminder is due.

The Display Icon check box controls whether a small icon (sicn), alternating with the Apple icon in the Apple menu (or the Application menu in System 7), will be displayed when a reminder is due. AMReminder will wait for the user to acknowledge the notification by selecting the Cancel Reminder command in the Remind menu.

The next figure shows the appearance of a filled-in entry, and shows how the pop-up menu can be used to select a sound to be played when the notification occurs. Operation of the pop-up menu is provided by code automatically generated by AppMaker; AppMaker even generated code to play the selected sound.



AMReminder

Add reminder for: Date: 8/13/91 Time: 10:00 AM [✓ PM]

Reminder: Backup Files

When reminding: [✓ Display icon in menu bar]
[✓ Display alert]
[✓ Play sound:] ✓Droplet

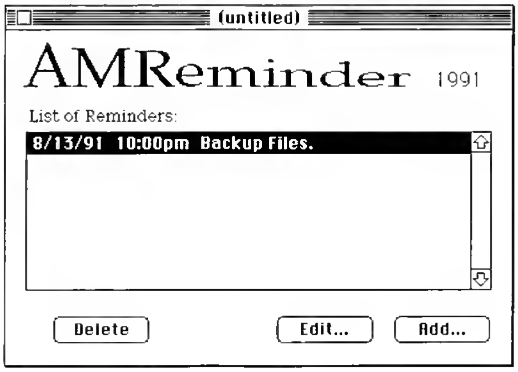
Dates are entered in the form mm/dd/yy and time is entered in hours and minutes (using a 12-hour clock), in the form hh:mm. A pair of radio buttons (labeled AM and PM) provides the ability to select a morning or evening time.

We have written our code so that the user must enter a date, a time, and a message. AppMaker generated code to enable or disable the OK button based upon a boolean condition. We wrote code to set the boolean variable true only if the date, time, and message fields were non-empty.

When an entry is complete, and the OK button is clicked, we have written code to use the information in the dialog entries to create an abbreviated display in the main window's scrolling list, as shown and described in the next section, "Editing Reminders."

Editing Reminders

When one or more reminders have been entered, they are displayed in abbreviated form in the main window. The figure below shows an entry that has been selected by clicking on the text of the entry in the list pane. In this figure the Edit and Delete buttons are active. This will be the case only when an entry has been selected.



The user can edit a reminder in one of two ways: by clicking in the list to select an entry, then clicking the Edit button; or by double-clicking an entry in the list. In either case, the entry will be displayed in its present form in the entry dialog, as shown previously. We added the code to implement these actions, both for double-clicking and for clicking the Edit button when an entry is already selected.

Any portion of the reminder can be changed at this point. Clicking Cancel in the entry dialog will cause any changes to be ignored. Clicking OK will cause the updated reminder to be displayed in the main window.

Reminders can be deleted as well as edited. To delete a reminder, the user must select it and click the Delete button in the main window. The entry will disappear and any entries below it in the list will scroll up to occupy the position of the deleted reminder. Notice that when an entry is deleted, the highlight is removed and the Edit and Delete buttons are deactivated. This behavior is produced by code that we added to the application.

We must also accommodate any actions the user takes that will require saving the current reminders. If the user clicks the close box in the main window, the application must ask the user if he wishes to save the file. The application should follow similar logic when the Close option in the File menu is chosen. The user can also choose either the Save or Save As options, to save the list of reminders.

Completing the Application

In the previous sections, we have discussed the various user interface elements and the operation of the AMReminder application. This section is intended to recap the portions of the application that are automatically generated and those that we have added.

AppMaker-Generated Code

AppMaker will automatically generate code to display most of the user interface elements in the AMReminder application. This includes all the menus, all items in the main window, and all items in the entry dialog.

The code for displaying the entry dialog is automatically generated; however it is not automatically called by the generated program. This action occurs when the Add or Edit buttons are clicked, or when an existing entry is double-clicked. Because AppMaker doesn't know that this functionality is required, it can not automatically generate code to show the dialog in those cases. It was a simple matter for us to add this capability.

Code is also generated for recognizing each event (keystroke, mouse click, etc.), and the generated code contains comments indicating where code should be added to handle these events.

Our Additional Code

The generated code provides a substantial skeleton program, in which many features are completely implemented. The additional code that we had to write includes the following:

- Reading and writing reminder entries from or to a file, when the user chooses the Open and Save commands from the File menu
- Implementing the appropriate actions when the user clicks the Add, Edit, and Delete buttons in the main window

- Activating and deactivating the Edit and Delete buttons, when appropriate
- Enabling and disabling menu items, when appropriate
- Verifying that reminder entries have been properly entered
- Formatting abbreviated entries and displaying these in the list pane in the main window
- Saving the completed reminder entries, and accessing those entries when needed
- Periodically checking to see if a notification should be posted, and making use of the Notification Manager

The amount of code required to implement the above items is a small portion of the total amount of application code. Most of the code is either automatically generated or is present in AppMaker's library or in the selected class library, depending on the language being used.

The modules that have been automatically generated for the AMReminder application are going to be very similar to those in the majority of other Macintosh applications. All applications will have code to enable or disable their menu items. All will have menu commands that require handling. In many cases, additional windows or dialogs can be handled in a fashion similar to our entry dialog. And events related to buttons and other interface elements will often be handled much the same as in our example application.

Studying AppMaker's generated code can prove to be extremely valuable, particularly when learning a new language or class library.

Chapter 5
Programming a Procedural Application

This chapter describes the code that implements the example AMReminder application's user interface that was described in Chapter 4. The language used for this particular implementation is procedural (as opposed to object-oriented) THINK Pascal. Although THINK Pascal has a substantial set of object-oriented programming features, this chapter uses pure procedural programming methods for the implementation of the example application.

The information in this chapter, although presented in THINK Pascal, applies to all of the procedural languages AppMaker supports: THINK Pascal, THINK C, MPW Pascal, MPW C, A/UX C, and FORTRAN. (FORTRAN is supported by an add-on product, available separately from Bowers Development.)

As with any application , the user interface must first be designed. AppMaker is quite useful during this portion of the project. All the visible elements of AMReminder's user interface were drawn using AppMaker's built-in tools. This includes all the menus, the main window and its constituent buttons and scrolling list, as well as the entry dialog and its buttons, check boxes, radio buttons, static text, and text edit panes.

Once the user interface has been designed and its features are satisfactory, then AppMaker will generate the source code to implement the display and operation of the user interface elements.

The remainder of this chapter will discuss the generated code in terms of its structure and functionality. We will present a tour of the code, providing insight into how AppMaker generates code for any procedural application.

While reading this chapter, you will find it helpful to open the source code files for AMReminder in the Examples:Procedural folder and view or print them.

AppMaker's Code Generation Scheme

AppMaker has a very flexible code generation scheme that is controlled by **template resources** that are part of the AppMaker application. Each target language in AppMaker's repertoire has a set of associated templates that direct its code-generation task. In the case of procedural THINK Pascal, the 'TmpP' resources are used to direct code generation.

AppMaker interprets the directives in the 'TmpP' resources, to produce code that is specific to a particular user interface design. For more information, see the description of the template language in Appendix D, "Customizing AppMaker's Code Generator."

Code Templates

In this section we will describe how AppMaker generates the code for our AMReminder example application, by using AppMaker's 'TmpP' resources as a guide. Each resource of this type contains a template language description of what to generate for its associated user interface element. The following table lists the major template resources:

Resource Name	Code Generated
GenWhat	Defines what to generate
---BuildOrder	---BuildOrder for THINK Pascal
Globals	Globals file
ResourceDefs	Resource definitions file
Window	Source file for a window
MainMenu	Code to manage menus
Menu	Source file for a menu
ModalDialog	Source file for modal dialog
ModalOneShot	Source file for palette dialog
ModelessDialog	Source file for modeless dialog
Dispatcher	Dispatcher file

Resource Name	Code Generated
Menu.File	Source file for File menu
Menu.Font	Source file for Font menu
EachWindow	Code fragments for windows
EachDialog	Code fragments for dialogs
EachMenu	Code fragments for menus
EachMenuItem	Code fragments for menu commands
DefineResources	Code to define resources
EachResource	Code fragments for resources
DoMenuItem	Menu commands
DoMenuItem.File	Menu commands in File menu
DoMenuItem.Edit	Menu commands in Edit menu
MakeFile	'.make file' for MPW
Main	The main entry point
Data	Application-specific data access module

Note that the 'TmpP' resources are applicable to both the MPW and THINK compiler environments.Where these two environments differ, directives in the templates generate environment-specific code.

The key to the entire code generation process is AppMaker's interpretation of the first template: GenWhat. This template establishes the top-level structure of a procedural Pascal program. The content of the GenWhat template is shown below:

```
%if lang = MPW%
   %genfile MakeFile appfilename+.make%
%elsif lang = Think%
   %genfile ---BuildOrder ---BuildOrder%
%endif%
%genfile Main appName+Main.p%
%genfile Globals Globals+.p%
%for each dialog gen sourcefile%
%for each window gen sourcefile%
%genfile Dispatcher Dispatcher+.p%
%for each menu gen sourcefile%
%genfile MainMenu MainMenu+.p%
%if not fileExists appName+Data.p%
   %genfile Data appName+Data.p%
%endif%
%genfile ResourceDefs ResourceDefs+.p%
```

Notice that code generation begins with the creation of either an MPW Pascal Makefile, or a ---BuildOrder file for THINK Pascal. Following this, code generation proceeds as described below:

- ☐ The Main file is generated with a filename that is the application's name with 'Main' appended.
- ☐ The Globals file is generated.
- ☐ For each dialog defined in the user interface design, AppMaker generates a source file. In the case of the AMReminder example, a modal dialog file called Add is generated.
- ☐ For each window defined in the user interface design, AppMaker generates a source file. In the case of the AMReminder example, a file called MainWindow is generated.
- ☐ The Dispatcher file is generated.
- ☐ For each menu defined in the user interface design, AppMaker generates a source file. In the case of the AMReminder example, files called FileM, EditM, and RemindM are generated.
- ☐ The MainMenu file is generated.
- ☐ Unless it already exists, the Data file is generated with a filename that is the application's name with 'Data' appended.

☐ The ResourceDefs file is generated.

Each of the templates that causes a file to be generated also references other templates, which generate custom code according to the user interface definition. In particular, each user interface item (button, check box, radio button, etc.) requires code to create the item in a window or dialog box, to initialize the item, dispose of the item, and to handle mouse clicks or other events. The code for the individual items is generated by template resources from the following table:

Resource Name	Code Generated
Button	Button items
Checkbox	Check box items
RadioButton	Radio button items
StatText	Static text items
EditText	Edit text items
EditNum	Numeric edit text items
Icon	Icon items
Picture	Picture items
Line	Line items
Rect	Rect items
Palette	Palette items
Popup	Pop-up menu items
List	List items
ScrollBar	Scroll bar items
Custom	Custom control items
UserItem	UserItem items
PictButton	Picture buttons
PictCheckbox	Picture check box items
PictRadioButton	Picture radio buttons
PictMulti	Multi-picture buttons
Slider	Slider items

Source Files for AMReminder

The AMReminder user interface, as described in Chapter 4, contains quite a few elements. This requires generation of a substantial amount of code. The source files automatically generated by AppMaker are listed below:

- **BuildOrder**—establishes the order in which the files should be listed in the THINK Pascal project window.
- **AMReminderMain**—the starting point for the generated application.
- **Globals**—defines a number of global symbols, a window information record that contains fields associated with the main window in our application, and a set of procedures for initializing the window record fields as well as selecting the record for the active window.
- **Add**—code for drawing the new reminder entry dialog box, as well as responding to specific events associated with its items (Date, Time, and Message entry fields, as well as the radio buttons, check boxes, and Play Sound pop-up menu).
- **MainWindow**—code for drawing the main window, as well as responding to specific events associated with its items (scrolling list, Add, Edit, and Delete buttons).
- **Dispatcher**—routines called by the main event loop, that route events to the specific window in which they occur.
- **FileM**—handles most of the standard File menu commands, such as New, Open, Save, Save As, and the like.
- **EditM**—handles all the standard Edit menu commands, including Cut, Copy, Paste, Clear, and Select All.
- **RemindM**—handles commands in the Remind menu.
- **MainMenu**—loads the initial menu bar, handles the Apple menu, and updates the menus and their individual items.
- **AMReminderData**—an application-specific module for accessing the files and data structures of your application.
- **ResourceDefs**—contains a set of global constant and variable resource declarations.

Library Files for AMReminder

AppMaker also provides a number of library routines, in source form, that are referenced by the generated code. The library files include the following:

- **EventLoop**—recognizes and responds to all the Macintosh events. It also contains the code to initialize the toolbox.
- **AEvent**—handles AppleEvents for opening the application, opening documents, and quitting.
- **Windowing**—handles window events, such as mouse clicks in the content, drag, grow box, go-away box, and zoom box regions of the window.
- **DialogAids**—handles numerous chores for dialog items, such as outlining the default button, checking or unchecking check boxes, enabling or disabling items, etc.
- **WindowAids**—handles numerous chores for typical window items, such as plotting icons, drawing pictures, tracking button clicks, hilighting scroll bars, etc.
- **Cursors**—contains routines for setting the cursor shape, such as ShapeCursor, SetBusyCursor, and SpinCursor.
- **DoScrap**—contains routines to cut and paste text via the Clipboard.
- **Scrolling**—handles all the standard actions required to scroll windows.
- **Miscellany**—contains miscellaneous routines.

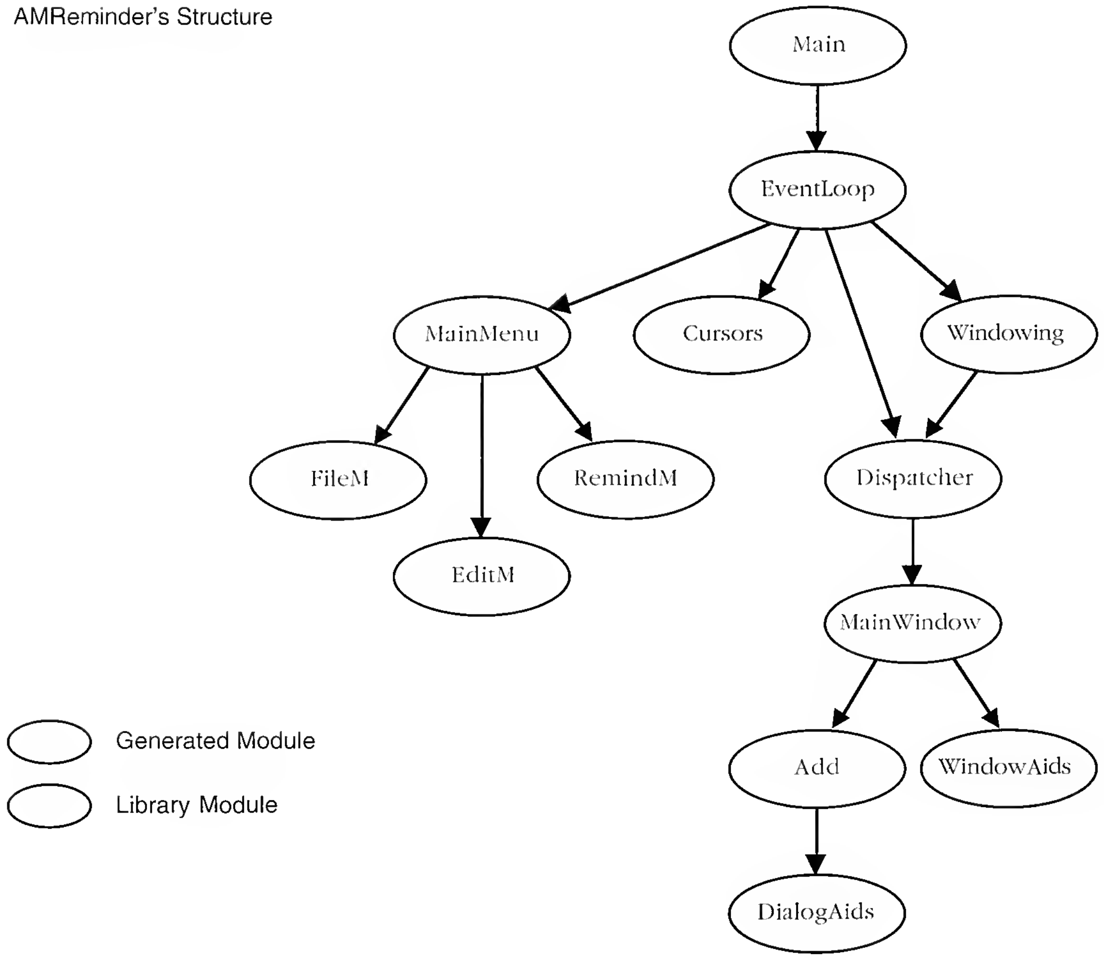
Analysis of AMReminder's Structure

While it is useful to know the type of code that is generated for a particular design, it is also necessary to know the structure of the program as a whole before adding the refinements that will implement its full functionality.

The figure on the next page illustrates the basic program structure, in terms of its components and the flow of events and messages between them. In the figure, the shaded ovals correspond to AppMaker library files, and the white ovals correspond to generated source files.

Not all the program's connections are shown in the figure. It is mainly intended to stress the major elements in the program's structure and provide insight into where to look to begin adding code to complete the program's functionality.

AMReminder's Structure



The flow of the program begins in the AMReminderMain file, which contains the main module. This routine calls the Initialize procedure in the EventLoop module, which loads the menus, loads the cursors, and initializes global variables, by calling routines in generated code and in other library modules. One of the routines Initialize calls is MainMenu's InitTitles routine, which in turn calls InitFileM, InitEditM, and InitRemindM. These provide a place for you to perform application-specific initialization.

The Main Loop

When the initialization is complete, the main routine calls the MainLoop procedure in the EventLoop module, which runs until the application's execution is terminated. The MainLoop procedure reads each event from the operating system's event queue, and dispatches these to other modules, as shown in the diagram. The MainLoop procedure is the focal point in any AppMaker-generated program. Once the event loop is entered, all subsequent program actions depend on what events are generated by the user.

Some events can be completely handled in an identical fashion for every program. For example, if the user drags a window, or clicks in its zoom box, AppMaker generates code to handle those events by calling routines in its Windowing library module.

If the user clicks in the content region of a window, then the Windowing module's DoContent routine calls the Dispatcher module. A Dispatcher routine routes the event to the appropriate module for the window in which the click occurred. In the case of the AMReminder application, clicks in the main window are dispatched to the MouseInMainWindow routine in the MainWindow module. It is in this routine that mouse clicks in the scrolling list are handled.

If the user clicks in one of the controls in a window, then the Windowing module calls the DoControl routine in the Dispatcher module, which in turn calls the ControlMainWindow procedure in the MainWindow module.

Most events are first handled in the EventLoop and Windowing modules for application-independent processing and then dispatched to appropriate routines in a window module for application-specific processing. Since an application can have multiple kinds of windows (e.g. a spreadsheet window and a graph window), the Dispatcher module examines the window's kind and then calls code for the particular window. Each generated window module has many event-handling routines:

- ☐ **OpenMainWindow**—Creates a new window from a WIND resource, creates controls and other items inside the window, and initializes window-related variables.
- ☐ **CloseMainWindow**—Closes the window, and disposes of items in the window.

- **ControlMainWindow**—Handles clicks in the window's controls.
- **MouseInMainWindow**—Handles mouse-down events in non-control items in the window.
- **TypeInMainWindow**—Handles keystrokes when the window is the front window.
- **UpdateMainWindow**—Draws the contents of the window.
- **ActivateMainWindow**—Activates or deactivates items in the window when the window is activated or deactivated.
- **ResizeMainWindow**—Resizes items in the window, when the user resizes the window.
- **ScrollMainWindow**—Scrolls the contents of the window, when the user clicks or drags a scroll bar.

Most kinds of events are associated with a particular window. Update and Activate events specify a window; mouse-down events are routed to the window in which the mouse-down occurred; keystrokes are sent to the frontmost window.

Mouse-down events in the menu bar do not necessarily belong to a window. The EventLoop passes these to a routine in the MainMenu module, which in turn passes them to separate routines for each of the menus. Each menu module examines the item number for the event and calls a generated routine to handle a particular menu command.

When you add your code, you will usually add it to one of the window routines, or to a menu command routine.

The Window Information Record

Most Macintosh applications support opening multiple documents, and each document can have multiple window kinds (e.g. text windows and graphic windows). Each window needs its own set of handles for controls and other items, and needs access to the information it will display on the screen.

Rather than trying to store this data in global variables, AppMaker generates code to store each window's data in a separate Window Information Record (WinInfoRec), which is allocated on the heap. A pointer to the WinInfoRec is stored in the refCon field of each WindowRecord.

```
type
   WinInfoRec          = record
    {Standard fields:}
      text:            TEHandle;
      vScroll:         ControlHandle;
      hScroll:         ControlHandle;
      fileNum:         integer;
      volNum:          integer;
      dirty:           boolean;
      filename:        StringHandle;
      windowKind:      (noWindow,
                        WMainWindow, fillerWK);
      witlHandle:      Handle;
          {Window itemlist resource}
      wictHandle:      Handle;
          {Window item color table resource}

    {Application-specific fields:}
     {for MainWindow:}
      AddHandle:       ControlHandle;
      EditHandle:      ControlHandle;
      EditEnabled:     Boolean;
      DeleteHandle:    ControlHandle;
      DeleteEnabled:   Boolean;
      List7Handle:     ListHandle;
      List7Choice:     integer;
   end; {WinInfoRec}
   WinInfoPtr          = ^WinInfoRec;
```

Several of the components of this record are standard for all windows, while others are specific to the user interface design for a particular window. For AMReminder, the AddHandle, EditHandle, DeleteHandle, and List7Handle define handles to the three buttons, with corresponding names, and the scrolling list in the window.

WinInfoRec is defined in the Globals module. A pointer to the current window's information record is stored in the global variable, 'cur'. AppMaker library code and generated code maintains and accesses the value of 'cur'. Your code will access 'cur' from many places but won't modify it directly. In cases where you need to refer to another window's data, you can call SetInfo to set 'cur' to point to another window's information record.

Standard variables in the Globals module include:

- **cur**—points to the current window's WinInfoRec
- **curWindow**—points to the current window's Macintosh WindowRecord
- **curEvent**—holds the latest event record
- **sysConfig**—has fields to identify what operating system features (e.g. Color QuickDraw) are available
- **inBackground**—indicates whether the application is running in the background

Space for each new WinInfoRec is allocated and managed by a set of routines in the Globals module: SetNewInfo allocates a WinInfoRec when a window is opened; SetInfo places a window pointer into the curWindow variable, and a pointer to the corresponding instance of the WinInfoRec into the 'cur' variable; and DiscardInfo disposes the WinInfoRec when a window is closed.

The Data Access Module

Much of the code for reading and writing files can be application-independent, but most of it is very application-specific. AppMaker 1.5 generates a separate Data module to isolate the application-specific data structures and access routines.

The generated code for the File menu is mostly application-independent. It contains routines for opening, reading, writing, and closing files. This code calls routines in the Data module to perform physical and logical file I/O specific to your application.

Isolating the application-specific code into the Data module helps AppMaker generate better code in the File module and may also help you separate the details of data representation from the rest of your code.

AppMaker generates a skeleton Data module with comments to help you get started on the data structures and access routines for your application.

Completing AMReminder's Code

When the code for the AMReminder application is first compiled and executed, the main window, its controls, and all the menus will operate as intended. That is, when you click on the Add button, it will highlight, but no other action will take place. AppMaker doesn't know what you intend for this event to do, so no code is generated to handle the event. Similarly, if you pull down the Remind menu and choose the Hide Reminder command, the menu item will flash, as specified in your Control Panel settings, but nothing will happen. You must write code that implements the actions corresponding to events occurring in each of your user interface's items.

In completing the AMReminder application, we had to write code to implement the following functionality:

- Mouse clicks in the Add button in the main window must show the modal dialog for entry of a new reminder, and handle the fields and controls in the dialog.
- Mouse clicks in an active Edit button in the main window must display the entry dialog, with all the fields and settings that correspond to the highlighted reminder in the scrolling list.
- A double-click on an entry in the scrolling list in the main window must perform the same function as highlighting the entry, and clicking the Edit button.
- Mouse clicks in an active Delete button in the main window must delete the highlighted reminder in the scrolling list.
- All the menus and their items must be enabled or disabled appropriately whenever an event is processed.
- Reminder entries must be written to a disk file for future use at the user's discretion.

Each of the above items will be discussed in the following sections. Most of our additions were in the Data module or the MainWindow module. All changes from the generated code are marked with comments in the source files in the Examples:Procedural folder.

Making Reminder Entries

A reminder entry is stored in memory in a record called a ReminderRec, which we defined in the Data module. We decided to maintain all of the reminder entries for a document in a linked list. We added a handle to this linked list to the WinInfoRec. The AppMaker library provides routines for managing the linked list.

The Data module contains two categories of routines: those that manipulate internal data structures so that the rest of the program need not be concerned with the detailed representation of the data; and those that perform physical I/O between external files and the internal data structures.

The skeleton Data module generated by AppMaker had both categories of routines. We replaced the generated data access routines with AMReminder-specific routines to add, get, put, and delete reminder entries from the linked list. We modified the I/O routines to read reminder entries from a file, to write them to a file, and to remove them from memory when the file was closed.

The Add button lets the user add a new reminder. When the user clicks this button, the ControlMainWindow routine calls the DoAddButton routine.

The DoAddButton procedure initializes the fields of an AddRec to default values for a new reminder, then calls a function named GetAdd, which can be found in the Add module. This function returns a true value if the user clicked the OK button in the entry dialog; otherwise, it returns a false value.

Enabling the Buttons

The Edit and Delete buttons should be enabled if and only if there is a reminder selected in MainWindow's list of reminders. We wrote a simple procedure to do this. UpdateButtons calls the library routine GetListChoice to determine whether a list row is selected, then calls another library routine, EnableControl, to enable or disable the Edit and Delete buttons accordingly.

By the way, most of the library routines contain just a few lines of toolbox calls. We could have called the toolbox routines LGetCell and HiliteControl directly. Calling the library routines made our UpdateButtons routine shorter, simpler, and more readable.

We called UpdateButtons from other places in the MainWindow which could affect whether a reminder was selected. In particular, we called it from MouseInMainWindow if the user clicked in the list.

Managing the Add Dialog

We're going to take a slight diversion at this point to show you how the entry dialog (which was named Add when it was defined in AppMaker, to correspond with the Add button) is managed. We will conclude our discussion of the procedures in the MainWindow module, when we return.

The GetAdd function is located in the Add module. It is a boolean function which takes an AddRec as a parameter. It displays the Add dialog, using the values in the AddRec as initial settings. After interacting with the user to change the AddRec values, GetAdd returns either true or false, depending on whether the user clicks its OK or Cancel button, respectively.

The GetAdd function was generated by AppMaker, but we have added a couple of lines of code to enable the OK button, when appropriate. All the other code is used as generated by AppMaker.

The first section of code in the GetAdd function sets each of the items in the Add dialog to its corresponding value in the AddRec that was passed to the function. Immediately following this initialization step, the generated code enters its main loop, which continues looping until the boolean variable called 'done' becomes true.

In each pass through the loop, the OK button is either enabled or disabled. In order for it to be enabled, the Date, Time, and Message edit text items must contain entries. AppMaker generated code to enable the OK button based on a boolean condition. We wrote code to set the boolean variable, based on checking the content of these fields.

After the state of the OK button is set, the function calls ModalDialog to handle the events generated by the user's actions. When each event occurs, ModalDialog returns its item number and the generated code handles the event associated with that item. The remainder of the function consists of a big case statement, with a case for each of the dialog box's items. When an event for an item is generated, the appropriate case to handle the item is executed.

If either the OK button logic or the Cancel button logic sets the 'done' variable to true, the loop will terminate and the dialog will be disposed. The result returned to the caller will either be true or false, depending on whether OK or Cancel was clicked.

Editing a Reminder Entry

Reminder entries can be edited either by double-clicking on a reminder, or clicking once on the entry to highlight it, then clicking the Edit button in the main window. The single and double-click events are handled by the MouseInMainWindow routine in the MainWindow module, and the event resulting from clicking the Edit button is handled by the ControlMainWindow routine.

When either of the above events occurs, the routine DoEditButton is called to handle re-opening the Add dialog window, with the items set to the values associated with the selected entry. Handling of the Add dialog is accomplished with the same GetAdd function described earlier. Setup of the dialog fields is handled in the DoEditButton routine.

This procedure calls the library routine GetListChoice to determine which reminder is selected, then calls the Data modules's GetReminder to get the ReminderRec for the selected entry. DoEditButton copies the current reminder entry to a local AddRec, which it passes to the GetAdd function. When GetAdd returns with a true result, DoEditButton updates the ReminderRec from the modified AddRec and calls the Data module's PutReminder to update the visible list in the MainWindow. If GetAdd returns false (indicating that the Cancel button was clicked), then nothing is done with the contents of the AddRec.

Although the complete reminder entry is saved in the linked list, an abbreviated version, consisting of the Date, Time, an 'am' or 'pm' indicator, and the contents of the Message fields, is displayed in the main window.

Deleting a Reminder Entry

Delete

A reminder entry is deleted by selecting it in the main window, to highlight the entry, then clicking the Delete button. The code that handles these events is found in the MainWindow module. The mouse click that selects the entry is processed by the MouseInMainWindow routine and the event that results from clicking the Delete button is handled by the ControlMainWindow routine.

When the Delete button is clicked, the ControlMainWindow procedure calls the DoDeleteButton routine, which is generated as an empty placeholder by AppMaker, and which we filled in to complete its functionality.

DoDeleteButton calls GetListChoice to determine which reminder is selected, then calls the Data module's DeleteReminder to remove it from the linked list and to update the visible list in the MainWindow. Since no reminder is now selected, DoDeleteButton calls UpdateButtons to disable the Edit and Delete buttons.

Handling AMReminder's Menus

The previous sections have discussed the main window, the entry dialog, and their items and controls. In this section, we are going to show how the application's menus are initialized and managed.

All menu events come from the EventLoop module and are passed into various routines in the MainMenu module. This module is responsible for initializing the menu bar, updating each menu item's display status, and dispatching commands to the module generated to handle each of the menus. There are three such modules: FileM, EditM, and RemindM, which correspond to the File, Edit, and Remind menus, respectively.

Modifying the MainMenu Module

The MainMenu module contains routines for initializing the other menu modules, for loading the menu resources, for handling Apple menu commands, and for dispatching other menu commands to their modules.

We didn't modify any of these routines. AppMaker generated everything we needed, including code to add 'snd ' resources for the Play Sound pop-up menu.

MainMenu also contains a routine for updating menus. After every non-null event, the MainLoop calls UpdateMenus to enable/disable menus and menu items and to place checkmarks where appropriate.

We modified UpdateMenus very slightly. We added two lines of code to enable or disable the Hide Reminders menu item. It is enabled if there is an open window or if there are any hidden windows (in which case the menu item reads Show Reminders, set elsewhere).

Modifying the Remind Menu Procedures

AppMaker generates a file named after each menu, and includes code for each of the menu's items. It also generates a general dispatcher routine called Do*menuName*. In the case of the Remind menu, it is DoRemind. This procedure, like the other menu dispatchers, contains a case statement that vectors the potential commands to the correct handlers. In the case of the Remind menu, there are only two items: Cancel Reminder and Hide Reminders. AppMaker generates code that calls the DoCancelReminder and DoHideReminders routines, when these items are selected, respectively. The code for DoCancelReminder has been left unmodified, as an empty routine.

The DoHideReminders code toggles the state of the window and the corresponding menu item. If the window is showing, then the menu item reads Hide Reminders. If the window is hidden, then the menu item reads Show Reminders. A global variable, hiddenWindow, contains a WindowPtr to the hidden window.

What's Missing in the AMReminder Application?

Up to this point, all the sections in this chapter have focused on the user interface features and how these are implemented—both with generated and additional code. The AMReminder application is still not complete, because there is more to do than simply implement the user interface.

For the purposes of this chapter, implementation of the remainder of the application's functionality will not be covered; however, we would be remiss by not describing what remains to be done, and where it should be added.

When a list of reminders has been entered into the main window, their records are also available in the linked-list, headed by the reminderList field that we added to the WinInfoRec. All the information necessary to determine whether it is time to call the Notification Manager to post a notification entry in its queue is available in these records.

The missing ingredient is a relatively simple routine that periodically compares the date and time for each of the reminder entries with the Macintosh's current date and time. When the current date and time equal or exceed the corresponding values for a reminder in the list, then the routine should call the Notification Manager's NMInstall routine to post a notification. The routine should also activate the Cancel Reminder item in the Remind menu. When the user acknowledges the notification by choosing the Cancel Reminder menu item, the routine should disable the menu choice, remove the reminder from the list, and call the Notification Manager's NMRemove routine to remove the entry from its queue.

The appropriate method for implementing the periodic routine that we need is to make use of the Idle events that the Macintosh's Event Manager generates when nothing else is going on. Idle events, like many other events are input by the MainLoop routine in the EventLoop module, which calls the Dispatcher's DoIdle routine. This procedure is empty in AppMaker's generated code; however this is the place where we should put the periodic routine that checks the linked list of reminders.

In our AMReminder application, about 95% of the code is concerned with managing the user interface and 5% to implement the underlying functionality. Of the 95%, over 80% has been generated automatically by AppMaker. We have had to add the remaining 20% to activate the user interface, which is a small percentage of the overall code.

AppMaker Library for Procedural Applications

The AppMaker product includes a set of library routines for each language and its variants. The support routines for procedural applications are almost identical in implementation for THINK C and THINK Pascal, and for MPW C and MPW Pascal. The only differences are small variations to support the vagaries of the individual languages.

In all major respects the routines in these modules, regardless of the language, are functionally equivalent, and have identical calling sequences. In the sections that follow, the interface and description of each of the library routines will be presented. The interface will be presented in Pascal terms, but the conversion of calling sequences from Pascal to C is a rather straightforward process, and one that should be familiar to C programmers.

EventLoop Routines

The EventLoop module is the center of control for all AppMaker-generated procedural applications. This module contains a number of important routines, as well as the main logic to manage the acquisition and dispatching of all the events that occur during the application's execution.

```
Procedure Initialize;
```

Responsible for initializing all the toolbox managers, determining the machine environment in which the application is running, loading the menu bar and cursors, and initializing the desk scrap. It also calls the generated Dispatcher module to initialize any modeless dialogs and call the initialization routine for each defined menu.

```
Procedure MainLoop;
```

Contains the main event loop for all AppMaker-generated procedural applications. It calls a number of internal routines, which interface with the Dispatcher, MainMenu, and Windowing modules. None of the internal routines is callable from other modules.

You may want to examine these routines, however, because they lend insight into the flow of events in a generated application. In general, mouse clicks are dispatched to the Windowing module, for further sorting out. Keystrokes, update, activate, and idle events are sent directly to appropriate routines in the Dispatcher module. Other events, such as disk insertions and suspend or resume events are handled within the EventLoop module's private routines.

The UpdateMenus procedure, in the MainMenu module, is called by the MainLoop procedure every time an event is processed. In addition, each time through the event loop, the ShapeCursor procedure, in the Cursors module is called to draw the appropriate cursor shape.

AEvent Routines

The AEvent module contains routines to respond to AppleEvents received by your program. It handles the required AppleEvents kAEOpenApplication, kAEOpenDocuments, kAEPrintDocuments, and kAEQuitApplication. If your program supports additional AppleEvents, you might create your own module using AEvent as a model.

`Procedure InitializeAE;`

Calls AEInstallEventHandler to create entries in the AppleEvent dispatch table for the required AppleEvents, pairing them with the corresponding AppleEvent handlers DoAEOpenApp, DoAEOpenDoc, DoAEPrintDoc, and DoAEQuit. InitializeAE is called by the library during program initialization (if AppleEvents are available), so your code should not need to call this procedure.

`Procedure DoHighLevelEvent;`

Calls AEProcessAppleEvent, which looks in the AppleEvent dispatch table for an entry that matches the AppleEvent and, if it finds one, calls the corresponding AppleEvent handler installed by InitalizeAE. DoHighLevelEvent is called by the main event loop when high-level events occur, so your code should not need to call this procedure.

```
Function   MyGotRequiredParams(theEvent:AppleEvent): OSErr;
```

Used by the AppleEvent handlers (below) to verify that all of the AppleEvent's parameters have been processed. If no unprocessed parameters are found then it returns noErr, otherwise it returns errAEEventNotHandled. If you write your own AppleEvent handlers, they should probably also call this procedure.

```
Function   DoAEOpenApp   (theEvent:     AppleEvent;
                          reply:        AppleEvent;
                          refCon:       longint): OSErr;
```

The AppleEvent handler for kAEOpenApplication. It calls the generated procedure OpenApp in the FileM module.

```
Function   DoAEOpenDoc   (theEvent:     AppleEvent;
                          reply:        AppleEvent;
                          refCon:       longint): OSErr;
```

The AppleEvent handler for kAEOpenDocuments. For each filename in the AppleEvent it calls the generated procedure OpenDoc in the FileM module.

```
Function   DoAEPrintDoc  (theEvent:     AppleEvent;
                          reply:        AppleEvent;
                          refCon:       longint): OSErr;
```

The AppleEvent handler for kAEPrintDocuments.

```
Function   DoAEQuit      (theEvent:     AppleEvent;
                          reply:        AppleEvent;
                          refCon:       longint): OSErr;
```

The AppleEvent handler for kAEOpenApplication. It calls the generated procedure DoQuit in the FileM module.

Windowing Routines

The Windowing module receives all of the mouse click events, when these are detected in the main event loop. The routines in this module can handle most of the mouse click events. Those that can't be handled are passed to appropriate routines in the Dispatcher module.

`Procedure DoContent` `(whichWindow: WindowPtr);`

Handles mouse clicks in the content region of a window. The following variations of this event are handled:

- A click in a window that is not the active front window will select that window.
- A click in a text field will result in a call to the TEClick toolbox routine.
- A click in one of the scroll bars is handled by calling the TrackScroll routine in the Scrolling module.
- A click in a control in the window is passed on to the DoControl routine in the Dispatcher module.
- A click in the content region, where none of the other conditions applies results in a call to the MouseInContent routine in the Dispatcher module.

`Procedure DoDrag` `(whichWindow: WindowPtr);`

Handles the situation where the user has elected to drag the window by its title bar to another screen location. Your code should not need to call this procedure.

`Procedure DoGrow` `(whichWindow: WindowPtr);`

Handles resizing a window when the user clicks and drags in the grow box. Your code should not need to call this procedure.

`Procedure DoGoAway` `(whichWindow: WindowPtr);`

Handles closing a window when its go-away box has been clicked. It calls the DoClose routine in the generated FileM module. Your code should not need to call this procedure.

```
Procedure DoZoom      (whichWindow:  WindowPtr;
                       inOrOut:      integer);
```

Handles zooming the window, in or out, depending on its current state. Your code should not need to call this procedure.

WindowAids Routines

The WindowAids module has a large number of procedures and functions that perform services for events that occur in various types of window items.

```
Function  GetWRect    (itemNr:       integer): Rect;
```

Returns the rectangle for an item in a window.

```
Procedure SetWFont    (itemNr:       integer);
```

Sets the text font, size, style, and text transfer mode for the current port from item itemNr in the current window; in generated code, this is usually part of preparing to draw the item. Also sets the WindowAids module variable textJust to the item's justification setting.

```
Function  GetWindow   (windowID:     integer): WindowPtr;
```

Creates a new frontmost window from WIND resource id windowID. If Color Quickdraw is present, create a color window, otherwise create a black and white window.

```
Function  NewV1List   (bounds:       Rect;
                       parentWindow: WindowPtr): ListHandle;
```

Creates a new single-column list in a window. It is set up to contain simple text items, and has a cell-size of the full width of the pane that was defined in the interface. It has a vertical scroll bar, but no horizontal scrolling capabilities. The function returns a handle to the new list. The list does not get drawn until your program calls LDoDraw.

```
Function  GetListChoice(var choice:  integer;
                       list:         ListHandle): boolean;
```

If any cells are selected in the scrolling list whose handle is passed in the list parameter, sets choice to the 0-based index of the first selected cell and returns true. If no cells are selected, sets choice to -1 and returns false.

```
Procedure SetListChoice(choice:       integer;
                        list:         ListHandle);
```

Selects a cell in a scrolling list. choice is 0-based. Does not deselect any other cells.

```
Procedure GetListRow   (var data:     Str255;
                        index:        integer;
                        list:         ListHandle);
```

From a scrolling list of text items, returns the string associated with a given cell. index is 0-based.

```
Procedure SetListRow   (data:         Str255;
                        index:        integer;
                        list:         ListHandle);
```

In a scrolling list of text items, sets the string associated with a given cell. index is 0-based.

```
Procedure AddToList    (data:         Str255;
                        list:         ListHandle);
```

Creates a new row in the list pointed to by the list parameter, and sets its contents to the string specified in the data parameter.

```
Procedure DrawList     (list:         ListHandle);
```

Draws the frame for the list specified by the list parameter, then calls the toolbox LUpdate procedure to draw the contents of the list within the frame.

```
Procedure TextIDBox    (textID:       integer;
                        bounds:       Rect);
```

Loads a TEXT resource, specified by the textID parameter, then draws the text string inside the rectangle specified by the bounds parameter.

```
Procedure PlotIconID   (iconID:       integer;
                        bounds:       Rect);
```

Loads an ICON resource, specified by the iconID parameter, then plots the icon within the rectangle specified by the bounds parameter.

```
Procedure DrawPictureID(pictID:       integer;
                        bounds:       Rect);
```

Loads a PICT resource, specified by the pictID parameter, then draws the picture within the rectangle specified by the bounds parameter.

```
Procedure DrawGrayLine (bounds:       Rect);
```

Draws a gray line from the top-left coordinate to the bottom-right coordinate of the bounds rectangle.

```
Procedure UpdatePopup  (bounds:       Rect;
                        menuID:       integer;
                        choice:       integer);
```

Draws a pop-up menu, specified by the menuID parameter, with the item text specified by the choice parameter, inside a frame whose rectangle is specified by the bounds parameter.

```
Procedure TrackPopup   (bounds:       Rect;
                        menuID:       integer;
                        var choice:   integer);
```

Pops up a menu, specified by the menuID parameter, at the location specified by the bounds parameter, by calling the PopupMenuSelect toolbox routine. It then draws the selected item in the bounds rectangle.

```
Function  TrackButton  (button:       ControlHandle;
                        mousePos:     Point): boolean
```

Returns the result from calling the TrackControl toolbox routine, for the button pointed to by the button handle, passing the toolbox routine the initial mouse position given by the mousePos parameter.

```
Procedure TrackCheck   (checkBox:     ControlHandle;
                        mousePos:     Point;
                        var checked: boolean);
```

Tracks the mouse when it was originally clicked inside a check box, specified by the checkBox parameter, at the mouse position given by the mousePos parameter. If the TrackControl toolbox routine returns a non-zero value, the state of the checked parameter is toggled and the new state is written into the check box's value field.

```
Procedure TrackRadio    (radio:        ControlHandle;
                         mousePos:     Point;
                         var choice:   integer);
```

Tracks the mouse after it has been clicked on a radio button, specified by the choice parameter, at the mouse position specified by the mousePos parameter. It uses the value in the RefCon field of the control record to identify the group and the button in the group that is currently selected. If the mouse-up occurs while the mouse position is still within the original radio button, the currently selected button is deselected and the button specified by the choice parameter is selected.

```
Procedure TrackMulti    (multi:        ControlHandle;
                         mousePos:     Point;
                         var value:    integer);
```

Tracks the mouse after it has been clicked on a multi-picture button, specified by the multi parameter, at the mouse position specified by the mousePos parameter. If the TrackControl toolbox routine returns a non-zero value, the state of the multi-picture button is advanced to the next picture, wrapping around to the first picture if necessary.

```
Procedure TrackPalette (palette:      ControlHandle;
                        mousePos:     Point;
                        var choice:integer);
```

Tracks the mouse from the position specified in the mousePos parameter, in a MacPaint-style palette. It sets the choice parameter to the currently selected item in the palette. The choice parameter is an integer whose bounds are determined by the control's minimum and maximum value settings. These are usually between 1 and N.

```
Procedure EnableControl(scroll:       ControlHandle;
                        active:       boolean);
```

Enables or disables a control whose handle is passed in the scroll parameter. The control is enabled if the active parameter is true; otherwise, it is disabled.

```
Procedure HiliteScroll (scroll:       ControlHandle;
                        active:       boolean);
```

Uses the boolean value of the active parameter to set the specified control (usually a scroll bar) to active or inactive status. The scroll bar (or other control) is specified by the scroll parameter.

DialogAids Routines

There are a great number of routines in the DialogAids module that assist in handling all kinds of chores for various items in dialog boxes and windows. Many of these routines are automatically called by AppMaker's generated code. You can also call these routines, as desired. Most of these routines assume that the current port has been set to the dialog in which the item is located.

```
Function  GetDRect      (itemNr:        integer): Rect;
```

Returns the rectangle for an item in a dialog (more convenient than calling GetDItem).

```
Procedure LineItem      (dialog:        DialogPtr;
                         itemNr:        integer);
```

Draws a gray line from the top-left to the bottom-right points of the itemNr parameter in the dialog pointed to by the dialog parameter.

```
Procedure RectItem      (dialog:        DialogPtr;
                         itemNr:        integer);
```

Draws a rectangle around the itemNr item in the dialog pointed to by the dialog parameter.

```
Procedure SetUserItem   (itemNr:        integer;
                         DoDraw:        ProcPtr);
```

Stores a pointer to the procedure specified in the DoDraw parameter, for the item corresponding to the itemNr parameter.

```
Procedure OutlineButton(itemNr:         integer);
```

Outlines the default button in a dialog or window. The specific button is indexed by the itemNr parameter.

```
Procedure EnableDItem   (itemNr:        integer;
                         enable:        boolean);
```

Enables or disables an item in a dialog or window, using the itemNr parameter as its item number. If the enable parameter is true, the item is enabled; otherwise, it is disabled.

```
Procedure SetDText      (itemNr:       integer;
                         text:         Str255);
```

Sets the contents of an EditText or StaticText field to the string specified by the text parameter. The item number is contained in the itemNr parameter.

```
Procedure GetDText      (itemNr:       integer;
                         var text:     Str255);
```

Returns the text currently present in an EditText or StaticText field into the text parameter, for the item identified by the itemNr parameter.

```
Procedure SetDNum       (itemNr:       integer;
                         num:          longint);
```

Sets an EditText or StaticText field to the string equivalent of the long integer specified by the num parameter. The item to be set is specified by the itemNr parameter.

```
Procedure GetDNum       (itemNr:       integer;
                         var num:      longint);
```

Returns the numeric equivalent of the EditText or StaticText field addressed by the itemNr parameter. The value is stored in the num parameter.

```
Procedure SetCheckbox   (itemNr:       integer;
                         checked:      boolean);
```

Either draws or erases the X in a check box, depending on the value of the checked parameter. The check box item is addressed by the itemNr parameter.

```
Procedure DoCheckbox    (itemNr:       integer;
                         var checked:  boolean);
```

Toggles the current state of the check box identified by the itemNr parameter. If the current state of the parameter named checked is true, it is set to false and SetCheckbox is called with that value. The converse behavior will also occur. The state of the checked parameter is always toggled. The procedure is usually called in response to a click in a check box control.

```
Procedure SetRadio     (firstItem:    integer;
                        choice:       integer);
```

Sets the radio button identified by the choice parameter, in the group whose first item is identified by the firstItem parameter, to true. It assumes that the firstItem parameter refers to a dialog or window item, and that the choice parameter refers to the *n-th* item in that group. The choice parameter has a value from 1 to N.

```
Procedure DoRadio      (firstItem:    integer;
                        itemNr;       integer;
                        var choice:   integer);
```

Handles a group of radio buttons. It assumes that the choice parameter refers to the button (relative to the group) that is currently selected, that the firstItem parameter refers to the item number of the first radio button in the group, and that the itemNr parameter refers to the item that is to be selected. The doRadio procedure removes the highlight from the button identified by the choice parameter, sets the choice parameter to the relative item of the itemNr parameter, then sets the highlight of that item. The procedure is usually called in response to a click in a radio button control.

```
Procedure DoIconRadio  (firstIcon:    integer;
                        itemNr:       integer;
                        var choice:   integer);
```

Handles a group of icon radio buttons. It assumes that the choice parameter refers to the icon radio button (relative to the group) that is currently selected, that the firstItem parameter refers to the item number of the first icon radio button in the group, and that the itemNr parameter refers to the item that is to be selected. The DoIconRadio procedure removes the highlight from the icon radio button identified by the choice parameter, sets the choice parameter to the relative item of the itemNr parameter, then sets the highlight of that item. The procedure is usually called in response to a mouse click in an icon radio button.

```
Procedure SetCtlChoice (itemNr:       integer;
                        choice:       integer);
```

Sets the value of the control, identified by the itemNr parameter, to the value specified by the choice parameter.

```
Function  GetCtlChoice (itemNr:       integer): integer;
```

Returns the value associated with the control specified by the itemNr parameter.

```
Procedure DoPalette     (itemNr:       integer;
                         var choice:   integer);
```

Returns the value associated with the control whose item number is specified by the itemNr parameter. The procedure is usually called in response to a mouse click in a palette.

```
Procedure DoMultiState  (itemNr:       integer;
                         var value:    integer);
```

Handles a control that has multiple active states. The control is identified by the itemNr parameter, and the current value of the control is assumed to be stored in the value parameter. The routine determines if the value parameter is equal to the maximum value for the control, and sets the value parameter to the minimum value for the control. If the value parameter has any other value on entry to the procedure, the value parameter is increased by 1. After the value parameter has been updated, it is stored into the control. The procedure is usually called in response to a mouse click in a multistate control.

```
Procedure SetScrollItem(itemNr:        integer;
                        value:         integer;
                        min:           integer;
                        max:           integer;
                        pageSize:      integer);
```

Sets the parameters of a scroll item. The item is identified by the itemNr parameter. The minimum, maximum, and current values are set to the min, max, and value parameters, respectively. The pageSize parameter is stored into the reference constant for the control.

```
Procedure DoScrollItem  (itemNr:       integer;
                         var value:    integer);
```

Returns the value associated with the control specified by the itemNr parameter. The procedure is usually called in response to a mouse click in a scroll item.

```
Procedure DrawPopup     (itemNr:       integer;
                         menuID:       integer;
                         var choice:   integer);
```

Gets the item specified by the itemNr parameter and draws the Popup menu identified by the menuID parameter, with the choice specified by the choice parameter.

```
Procedure DoPopup       (itemNr:       integer;
                         menuID:       integer;
                         var choice:   integer);
```

Handles a Popup menu by calling the TrackPopup procedure in the WindowAids module. It tracks the mouse in the rectangle defined by the item specified by the itemNr parameter, for the menu specified by the menuID parameter, and the choice specified by the choice parameter.

```
Procedure InvertLabel   (itemNr:       integer);
```

Inverts the appearance of the label specified by the itemNr parameter. It is usually called in response to a mouse-down in a pop-up item.

```
Function  Vert1List     (itemNr:       integer): ListHandle;
```

Creates a vertical single-column list, with a vertical scroll bar and no horizontal scroll bar according to the specifications in the dialog item specified by the itemNr parameter. It uses the List Manager LNew toolbox function to establish the list, and returns its handle.

```
Function  FilterList    (var event:         EventRecord;
                         list:              ListHandle;
                         listitem:          integer;
                         dblClickItem:      integer;
                         var itemHit:       integer): boolean;
```

Called by a generated filter function to handle list-related events. It is doubtful that you will need to call this function. Code to do so will automatically be generated, when appropriate.

```
Function  FilterScroll  (var event:         EventRecord;
                         scrollItem:        integer;
                         ActionProc:        ProcPtr;
                         var itemHit:       integer): boolean;
```

Called by a generated filter function to handle mouse down events in scroll bar items. It is doubtful that you will need to call this function. Code to do so will automatically be generated, when appropriate.

```
Procedure FilterIcon    (event:        EventRecord;
                         firstIcon:    integer;
                         choice:       integer);
```

Called by a generated filter function to draw the currently selected icon. The other (unselected) icons in an icon radio group are drawn by the dialog manager. It is doubtful that you will need to call this procedure. Code to do so will automatically be generated, when appropriate.

```
Function  StandardFilter(whichDialog:      DialogPtr;
                        var event:         EventRecord;
                        var itemHit:       integer): boolean;
```

Called by a generated filter function after all the other filters have been called and have returned a false response. It outlines the default button, translates the Return and Enter keys to a click of the OK button, the Escape key to a click of the Cancel button, and handles the standard Command-X, Command-C, and Command-V keystrokes by calling DlgCut, DlgCopy, and DlgPaste, respectively. It is doubtful that you will need to call this function. Code to do so will automatically be generated, when appropriate.

```
Function  DoModalEvent (filterProc:   ModalFilterProcPtr): integer;
```

Processes the next modal dialog event and then returns. If the event is associated with an item in the current dialog, returns the item number, otherwise 0. It is called by MovableDialog, but it can also be called directly. Assumes that the frontmost window is a modal (or movable modal) dialog window.

```
Procedure MovableDialog(filterProc:   ModalFilterProcPtr;
                        var itemHit:  integer);
```

This procedure helps implement "movable modal" dialogs under System 7.0. (Under System 6.x, the movable modal dialog window frame is not implemented; the standard non-movable modal dialog frame is used instead.) It performs all the functions of ModalDialog, plus:

- ☐ Clicking and dragging in the dialog's title bar moves the dialog
- ☐ Application windows update behind the dialog

To generate a dialog using this routine, choose the "Movable" window frame in the Dialog Info dialog. To turn an existing modal dialog into a movable modal dialog, simply change the call to ModalDialog (Toolbox Dialog Manager) to MovableDialog (AppMaker library). Modifies global curEvent (Globals).

Cursors Routines

The Cursors module contains routines to control the mouse cursors used in your program. It handles the arrow, I-beam (text), and watch cursors. If your program supports additional cursors, you might copy this module into your program and enhance the LoadCursors and ShapeCursor routines.

```
Procedure LoadCursors;
```

Loads the cursor resources your program will need, initializing the Cursors module variables, and sets the cursor to the standard arrow cursor. It is called by the library during program initialization, so your code should not need to call this procedure.

```
Procedure ShapeCursor;
```

Sets the cursor according to the location of the mouse. If the mouse is over a text editing area, the cursor is set to the I-beam, otherwise the cursor is set to the standard arrow. It also sets the global cursorRgn, which enables the main event loop to determine if the mouse has moved enough to justify recomputing the cursor. It is called once each time through the main event loop, in order to keep the cursor as up-to-date as possible.

```
Procedure StartBusyCursor(aCurID:     integer);
```

This routine and the ones below implement a simple “spinning cursor” mechanism. They should be used when the user may have to wait for a time-consuming process. StartBusyCursor loads an animated-cursor resource (‘acur’) and prepares it for use. Since most programs only have one animated cursor, it can be called during program initialization. You MUST call StartBusyCursor before calling SpinCursor, SpinBackwards, or StopBusyCursor. These routines are not called by any generated or library code, but they are provided for your convenience.

```
Procedure SpinCursor;
```

After StartBusyCursor has been called, this procedure should be called as often as possible from your code during a time-consuming process. It will spin the cursor forward one frame (it actually only spins the cursor if enough time has gone by since it last spun the cursor, in order to keep the cursor spinning at an even rate).

`Procedure SpinBackwards;`

Similar to SpinCursor, except that it spins the cursor backwards.

`Procedure StopBusyCursor;`

Releases the memory allocated by StartBusyCursor.

DoScrap Routines

The DoScrap module contains routines to control the desk scrap (also known as the Clipboard). It is designed to handle text. If your program supports additional types of desk scrap data, you might copy this module into your program and enhance the ReadDeskScrap and WriteDeskScrap routines.

`Procedure ReadDeskScrap;`

Checks to see if the contents of the desk scrap have changed since the last call to InitScrap, ReadDeskScrap, or WriteDeskScrap. If the scrap has changed, it puts the type of the data in the scrap into the DoScrap module variable scrapType. If the scrap contains text, it also copies the text into the TextEdit internal scrap (for use by TEPaste). It is called by the main event loop when activate or resume events occur, so your code should not need to call this procedure.

`Procedure WriteDeskScrap;`

Checks the DoScrap module variable scrapDirty to see if there is text that needs to be copied from the TextEdit internal scrap to the desk scrap, usually because the user has performed Cut or Copy on some selected text. If there is such text, it is copied to the desk scrap. It is called by the main event loop when deactivate or suspend events occur, and before the application terminates, so your code should not need to call this procedure. The code generated to support the standard Cut and Copy items in the Edit menu automatically set scrapDirty, as well as the StandardFilter library routine which is called from all modal dialogs.

`Procedure InitScrap;`

Initializes the DoScrap module variables, and then calls ReadDeskScrap. It is called by the library during program initialization, so your code should not need to call this procedure.

Miscellany Routines

The functions and procedures in the Miscellany module are generally unrelated. They are miscellaneous routines that check for proper completion of I/O services, provide stop alerts, and perform various other functions.

```
Procedure Acknowledge  (alertID:     integer);
```

Displays a Stop Alert, using the alert resource specified by the alertID parameter. No action is taken, when the alert has been acknowledged. It is up to the calling procedure to determine what to do after the user acknowledges the message.

```
Function  Confirm      (alertID:     integer): boolean;
```

Displays a Caution Alert, using the alert resource specified by the alertID parameter. The function returns true if the user clicks the OK button; otherwise it returns false.

```
Function  CheckOS      (resultCode:  OSErr): boolean;
```

Tests the result of an OS call and posts an appropriate error message inside a Stop Alert indicating the reason for failure. If no appropriate message is found, the message: OS Error is displayed, along with the string equivalent of the result code. The function returns true if no error occurred; otherwise, false is returned.

```
Function  FileExists   (fName:       Str255;
                        vRefNum:     integer): boolean;
```

Calls the GetFInfo toolbox routine, using the fName and vRefNum parameters. It returns true if the specified file already exists; otherwise it returns false.

```
Function  CreateFile     (var sfInfo:   SFReply;
                          prompt:       Str255;
                          suggestion:   Str255;
                          creator:      OSType;
                          fileType:     OSType): boolean;
```

Creates a file whose location and name are specified by displaying the standard Save File Dialog to the user. The dialog will contain the prompt specified in the prompt parameter, and a suggested file name, specified in the suggestion parameter. If the file already exists, it is deleted. The created file is given the specified name, the type specified by the fileType parameter, and creator code specified by the creator parameter. The sfInfo parameter will contain the SFReply record.

```
Procedure ScaleWindow    (window:       WindowPtr;
                          scaleSize:    boolean);
```

Adjusts the size and position of a window relative to the size of the screen. It assumes that the current window size is appropriate for the original Macintosh screen size (512 x 342 pixels). It resizes the window in proportion to the size of the current screen. For example, a full size window on a Mac Classic becomes a full size window on a 19 inch monitor. The window is specified by the window parameter. If the scaleSize parameter is false, only the position of the new window is changed, not its size.

```
Procedure DrawClippedGrow   (x:         integer;
                             y:         integer);
```

Draws a grow icon, clipped to the coordinates specified in the x and y parameters if x or y is positive or zero, it is relative to the left or top of the window. If x or y is negative, it is relative to the right or bottom of the window. For example, a call of DrawClippedGrow(-15, 0) causes a grow icon to be drawn, clipped close to the right edge, but extending to the top edge of its position. This is appropriate for a window that has a vertical, but no horizontal scroll bar.

```
Procedure DoRadioMenu    (menu:         MenuHandle;
                          firstItem:    integer;
                          lastItem:     integer;
                          itemNr:       integer);
```

Used for menus which behave like radio buttons, when only one item can be checked. It unchecks all the items from the firstItem to the lastItem parameters, then checks the item specified by the itemNr parameter.

```
Procedure PlaySound     (soundsMenu:   MenuHandle;
                         itemNr:       integer);
```

Loads and plays the sound resource with the same name as the menu item with 1-based index itemNr in soundsMenu. The sound is played synchronously, that is, PlaySound plays the entire sound before returning.

The following routines implement a linked list of Items. These routines expect that the first field of every item is a Handle to the next item. They make no assumptions about the remainder of each item.

```
Procedure LinkLast      (var listHead: Handle;
                         newItem:      Handle);
```

Adds newItem to the end of the (singly) linked list whose last item (before calling LinkLast) is listHead.

```
Function  GetNth        (listHead:     Handle;
                         num:          integer): Handle;
```

Returns the num'th (1-based) item in the linked list whose last item is listHead, or nil if listHead is nil. Does not check if num is greater than the number of items in the list.

```
Function  UnlinkNth     (var listHead: Handle;
                         num:          integer): Handle;
```

Removes the num'th (1-based) item from the linked list and returns it, or nil if listHead is nil.

Scrolling Routines

The Scrolling module contains several routines that aid in scrolling-related functions. These are usually called by other routines, such as the DoZoom and DoGrow routines in the Windowing module; however they may be called when their use is indicated.

```
Procedure ResizeScrollBars;
```

Resizes either or both of the horizontal or vertical scroll bars, following resizing of the pane in which they are enclosed. In particular, it is called by DoZoom and DoGrow procedures.

```
Procedure DoScrollPart (whichScroll:      ControlHandle;
                        partCode:         integer);
```

When a mouse click occurs in a scroll bar, this procedure adjusts the scroll bar, identified by the whichScroll parameter, according to which part, specified by the partCode parameter, was clicked. A delta value is calculated by setting -1 if the up-arrow is clicked, +1 if the down-arrow is clicked, -pagesize if the click occurred in the page-up region, or +pagesize if it occurred in the page-down region. The pagesize value must have previously been stored in the control's refCon. The value associated with the scroll bar is adjusted according to the delta amount, and the scroll bar is redrawn.

```
Procedure TrackScroll  (whichScroll:      ControlHandle;
                        partCode:         integer;
                        where:            Point;
                        actionProc:       ProcPtr);
```

Handles mouse-down events in a scroll bar, specified by the whichScroll parameter. Depending on which part was clicked, as specified by the partCode parameter, and the mouse location, as specified by the where parameter, the routine calls the TrackControl toolbox routine to track the mouse. The procedure accepts an action procedure in its ActionProc parameter. If this procedure has been specified, it is called after tracking the mouse whose click originated in the scroll bar's Thumb, and within the TrackControl procedure for all other part codes. The TrackScroll procedure is called to handle clicks in horizontal or vertical scroll bars by the FilterScroll routine in the DialogAids module, and by the DoContent procedure in the Windowing module.

Dragging Routines

The Dragging module contains routines to support dragging in your program. These routines are not called by any generated or library code, but they are provided for your convenience.

```
Function StartMove     (var startPt:      Point;
                        var constraint:   integer): boolean;
```

If your program needs the ability to let the user click on an object and move it, call this function on mouseDown (with the mouse position in startPt) to implement such an interface. If the user moves the mouse more than 2 pixels in any direction and is still holding down the mouse button, it sets constraint (described below) and returns true (your program should then call TrackMove—see below). If the user has released the mouse button, it returns false. In either case, it changes startPt to the user's last mouse position, but no more than 2 pixels away from the original value of startPt. When returning true, constraint is set as follows: If curEvent.modifiers indicates that the user was not also holding down the option key, constraint is set to noConstraint. If the user was also holding down the option key and the user moved the mouse more horizontally than vertically, constraint is set to hAxisOnly; more vertically than horizontally, constraint is set to vAxisOnly.

```
Function TrackMove      (dragRgn:       RgnHandle;
                         startPt:       Point;
                         bounds:        Rect;
                         constraint:    integer;
                         var deltaH:    integer;
                         var deltaV:    integer;
                         actionProc:    ProcPtr): boolean;
```

After StartMove returns true, call this function to track the dragging (i.e., to have a dotted outline follow the mouse location until the mouse button is released). dragRgn is the region outline to drag. startPt is the mouse position (local coordinates). bounds is the overall area in which the dragging can take place. constraint is one of noConstraint, hAxisOnly, or vAxisOnly. Calls DragGrayRgn. When the mouse button is released, if the mouse location is within bounds, sets deltaH and deltaV to the distance moved from startPt and returns true, otherwise returns false and deltaH and deltaV may not contain useful values.

```
Function TrackRange     (anchorPt:      Point;
                         var range:     Rect): boolean;
```

If your program needs to have a rectangular "selection tool," call this function on mouseDown (with the mouse position in anchorPt) to implement such an interface. As the user moves the mouse, a dotted rectangle with one corner at anchorPt and the other at the mouse position will continually be drawn. When the user releases the mouse button, range is set to the last such rectangle. Returns true if this rectangle is non-empty, false if it is empty (i.e., the user returned the mouse to anchorPt).

```
Function TrackRect       (var sizeRect: Rect;
                          limitRect:    Rect;
                          actionProc:   ProcPtr): boolean;
```

If your program needs to be able to permit the user to create rectangular objects by clicking and dragging to indicate their size, as well as to resize such objects, call this function on mouseDown to implement such an interface. Pass the object's starting size in sizeRect. In limitRect, pass the rectangle that encloses the area in which the mouse is permitted to drag the object's lower-right corner; typically this is the area from the object's minimum permitted lower-right corner to its maximum permitted lower-right corner. If you want TrackRect to perform some action during dragging (such as displaying the value of sizeRect), pass a pointer to a procedure in actionProc, otherwise pass nil in actionProc. When the user releases the mouse button, if the mouse location is still within 20 pixels of any edge of limitRect, sets sizeRect to the new size and returns true, otherwise returns false and sizeRect is undefined.

```
Function TrackLine       (var sizeRect: Rect;
                          limitRect:    Rect;
                          actionProc:   ProcPtr): boolean;
```

Similar to TrackRect, except that instead of implementing the interface for creating or resizing rectangular objects, TrackLine implements the interface for creating or resizing lines.

ScrollText Routines

The ScrollText module contains routines to support scrolling text in your program. These routines are not called by any generated or library code, but they are provided for your convenience.

```
Procedure SetMaxScroll;
```

Examines the amount of text in the current text field of the current window (cur->text). If there is more than a pageful of text, recalculates the maximum value for the text's vertical scroll bar (if present). If there is less than a pageful of text, makes the vertical scroll bar inactive (if present). Typically called after the text has been edited or its window has beenresized, when its length (in lines) might have changed.

```
Procedure ScrollToBar;
```

Examines the vertical position to which the text in the current text field of the current window is scrolled, according to its TextEdit record (cur->text). If the TextEdit vertical position does not match the control value of the vertical scroll bar (if present), it scrolls the text until its vertical position matches the vertical scroll bar.

```
Procedure ScrollToSelection;
```

If the text in the current text field of the current window is scrolled so that the first character of selected text (or the insertion point, if no text is selected) is not visible, this procedure scrolls it into view, placing it vertically centered if possible.

```
Procedure ResizeText;
```

Should be called when a window containing nothing but text (and associated vertical and/or horizontal scroll bars) is resized. Invalidates the entire window, recalculates the rectangle in which the text is displayed, re-wordwraps the text, calls SetMaxScroll to recalculate the vertical scroll bar, calls ScrollToBar to make sure that the start of the selected text is visible.

```
Function  AutoScroll: boolean;
```

If the mouse position is above the text's rectangle and the text has a vertical scroll bar, scrolls both the text and its vertical scroll bar up by one line. If the mouse position is below the text's rectangle and the text has a vertical scroll bar, scrolls both the text and its vertical scroll bar down by one line. Should be called when the user has clicked in the text and dragged the mouse.

Chapter 6
Programming a THINK Class Library Application

This chapter describes the AMReminder example, as implemented by AppMaker for compilation by THINK C (Version 5.0) with the THINK Class Library. The generated code, the modifications, and the new code we added are in keeping with the object-oriented focus of the THINK Class Library (hereinafter called the TCL).

The information in this chapter, although presented in THINK C, applies equally to THINK Pascal with the TCL.

The visual user interface of the AMReminder application is identical to the descriptions and screen pictures shown in Chapter 4.

This chapter focuses on the organization of AppMaker's generated code for the AMReminder application, and also on the code we have added to complete the functionality of AMReminder's user interface. The fundamental difference between this and the procedural Pascal implementation of the AMReminder application (described in the previous chapter) is not the language we used, but the fact that the object-oriented features of THINK C and the TCL have been used throughout.

When reading this chapter, you will find it helpful to open the source code files for AMReminder in the Examples:TCL folder and view or print them.

AppMaker's Code Generation Scheme

As with procedural Pascal, or any of the other languages AppMaker supports, generation of code for THINK C with the TCL is governed by a set of template resources that specify the exact source code to implement and operate the user interface elements you define with AppMaker's visual tools. In this case, each of the templates is of type 'TmCT', which is stored in the resource fork of the AppMaker application.

AppMaker interprets the directives in the 'TmCT' resources, to produce code that is specific to the user interface that has just been designed. The directives form a **template language**, in which directives are enclosed by % characters, as described in Appendix D, "Customizing AppMaker's Code Generator."

Code Templates

In the sections that follow, we intend to describe the generated code for the THINK C and TCL version of the AMReminder application. The sequence of code generation and the content of each generated file is explicitly controlled by the template language in the 'TmCT' resources.

Although both THINK C and THINK Pascal use the THINK Class Library, the implementation of that library for each of these object languages is somewhat different. Therefore, a separate set of templates ('TmPT' resources) is provided to generate code for THINK Pascal with the TCL.

The following table lists most of the major code generation templates. These templates generate entire files of source code for each major interface element.

Resource Name	Code Generated
GenWhat	Defines what to generate
Main	The main entry point
App	Application subclass
zApp	Application superclass
Doc	Document subclass
zDoc	Document superclass
Window	Window subclass
zWindow	Window superclass
ModalDialog	Modal dialog subclass
ModalOneShot	Palette dialog subclass
ModelessDialog	Modeless dialog subclass
zDialog	Dialog superclass
ResourceDefs	Resource definitions file
Data	Application-specific data access module
EachWindow	Code fragments for windows
EachDialog	Code fragments for dialogs
EachMenu	Code fragments for menus
EachMenuItem	Code fragments for menu items
DefineResources	Code to define resources
EachResource	Code fragments for resources

The templates in the preceding table begin the generation of an entire source file, such as for a window or menu, and in the process will often refer to other templates to generate custom code according to the user interface design.

The following table contains a list of templates that are used to generate code for individual interface items such as radio buttons, check boxes, icons, and the like.

Resource Name	Code Generated
CButton	Button items
CCheckbox	Check box items
CRadioControl	Radio button items
CStaticText	Static text items
CEditText	Edit text items
CIntegerText	Numeric edit text items
CIconPane	Icon items
CPicture	Picture items
CGrayLine	Line items
CAMBorder	Rectangle items
Palette	Palette items
CStdPopupPane	Pop-up menu items
CArrayPane	List items
CScrollBar	Scroll bar items
Custom	Custom control items
CPane	User Items
PictButton	Picture button items
PictCheck	Picture check box items
PictRadio	Picture radio buttons
CMultiState	Multi-picture items
CSlider	Slider items
CLabeledGroup	Labeled group items
CScrollPane	Scroll Pane items

The templates listed in the preceding tables are specific to code generated for THINK C with the THINK Class Library. There is no equivalent set of templates for MPW C (however, separate templates are provided for procedural MPW C applications).

Source Files for AMReminder

When code is generated for applications that use the THINK Class Library (TCL), AppMaker takes a different approach to structuring the generated code. Because object-oriented programs have the ability to override methods from inherited superclasses, AppMaker generates two code files for each major element of the user interface (application, document, window, dialog, etc.).

The first file contains default methods for handling some of the commands or user actions that will not likely need to be changed. This file (and others like it) has a prefix character 'z' in its name. The second file, which is a subclass of the first, contains methods that are likely to contain code that is unique to the application, and do not have the character 'z' prefix. If, at a later date, AppMaker is invoked to make changes to the user interface, only the first set of files (the superclasses) are regenerated. This lets you continue to add features to the user interface and generate new code without fear that all your previous work has been lost.

AppMaker begins its code generation by interpreting the contents of the resource template called GenWhat, shown below:

```
%if not fileExists appname+Main.c%
   %genfile Main appname+Main.c%
%endif%
%if not fileExists appname+Data.c%
   %genfile Data appname+Data.c%
   %genfile Data.h appname+Data.h%
%endif%
%genfile zApp z+appname+App+.c%
%genfile zApp.h z+appname+App+.h%
%if not fileExists appname+App+.c%
   %genfile App appname+App+.c%
   %genfile App.h appname+App+.h%
%endif%
```

```
%genfile zDoc z+appname+Doc.c%
%genfile zDoc.h z+appname+Doc.h%
%if not fileExists appname+Doc+.c%
   %genfile Doc appname+Doc+.c%
   %genfile Doc.h appname+Doc+.h%
%endif%
%for each window gen sourcefile%
%for each dialog gen sourcefile%
%genfile ResourceDefs.h ResourceDefs.h%
```

As you can see, each AppMaker directive begins and ends with a % symbol. The word genfile indicates that a new file is to be generated. The template controlling the file to be generated, in this case, is called Main, and it is passed the argument appname, which is AppMaker's variable that holds the name of the application, with an extension of '.c'. See Appendix D, "Customizing AppMaker's Code Generator" for more information. The code generation proceeds as follows:

- After the main files for the application (in our case, AMReminderMain.c) is generated, then AppMaker generates the files corresponding to the Application superclass for our application, which is called zAMReminderApp.c. It will also generate a file called zAMReminderApp.h. The zAMReminderApp.c and zAMReminderApp.h files are always generated, unless you manually deselect them from AppMaker's list of files to generate.
- The next directive in the GenWhat template tests if a file called (in our case) AMReminderApp.c already exists. If so, the next directive is skipped; otherwise, the templates named App and App.h are interpreted, resulting in the generation of the AMReminderApp.c and AMReminderApp.h files. Once these are generated, AppMaker will not generate them again, ensuring that your changes and additions to these files are not overwritten when the interface is modified.
- The next directive causes the superclass document files (zAMReminderDoc.c and zAMReminderDoc.h) to always be generated. The following directive tests if the subclass files (AMReminderDoc.c and AMReminderDoc.h) are present, and if not, they are generated.

- At this point, AppMaker generates a file for each window defined in the user interface by interpreting the EachWindow template to create our zMainWindow.c and zMainWindow.h superclass files, and our MainWindow.c and MainWindow.h subclass files. If other windows had been defined, files would have been generated for these as well.
- AppMaker then generates a file for each dialog that has been defined, using its own methods and those inherited from the TCL's CDialog class. In our case, the dialog to enter a reminder is modal, so a set of files named zAdd.c, zAdd.h, Add.c and Add.h are generated.
- The final action in the GenWhat template is to generate a file called ResourceDefs.h. This header file contains #define directives for all of the resources defined by the user interface design.

Whenever one of the templates generates a source file, it will also refer to other templates that generate code for that source file. Appendix D, "Customizing AppMaker's Code Generator," contains examples that show this process in greater detail.

Library Files for AMReminder

AppMaker includes a large number of classes for use with the TCL. These classes enhance the capabilities of the TCL, providing additional functionality. The supplied Starter.π project references all of the standard TCL classes, as well as the additional AppMaker-supplied classes. Following is a summary of the purpose of each of the additional classes:

- ☐ **CAMArrayPane**—adds text styles to the TCL's CArrayPane class.
- ☐ **CAMBorder**—draws a rectangle (usually a border around other items.)
- ☐ **CAMButton**—adds text styles to button items.
- ☐ **CAMCheckBox**—adds text styles to check box labels.
- ☐ **CAMDialogDirector**—creates dialogs from DLOG resources.
- ☐ **CAMDialogText**—adds text styles to dialog EditText items.
- ☐ **CAMEditText**—adds text styles to EditText items.
- ☐ **CAMIntegerText**—adds text styles to CIntegerText items.
- ☐ **CAMPopupMenu** and **CAMPopupPane**—add font and size to pop-up menu items.
- ☐ **CAMRadioControl**—adds text styles to radio button items.
- ☐ **CAMStaticText**—adds text styles to static text items.
- ☐ **CAMStyleText**—adds text styles to CStyleText items.
- ☐ **CAMTable**—adds text styles to the TCL's CTable class.
- ☐ **CGrayLine**—implements a gray line item.
- ☐ **CLabeledGroup**—implements a rectangle and text label to visually group related items.
- ☐ **CMultiState**—implements multi-picture buttons.
- ☐ **CSlider**—implements a slider (scrollbar-like) control.
- ☐ **AMUtilities**—contains utility functions for setting text styles in window items.

Analysis of AMReminder's Structure

AppMaker's templates reveal quite a lot about the order and content of the generated code for our AMReminder application. By themselves, though, they do not provide much information on its structure, or the way in which commands and events flow through the code.

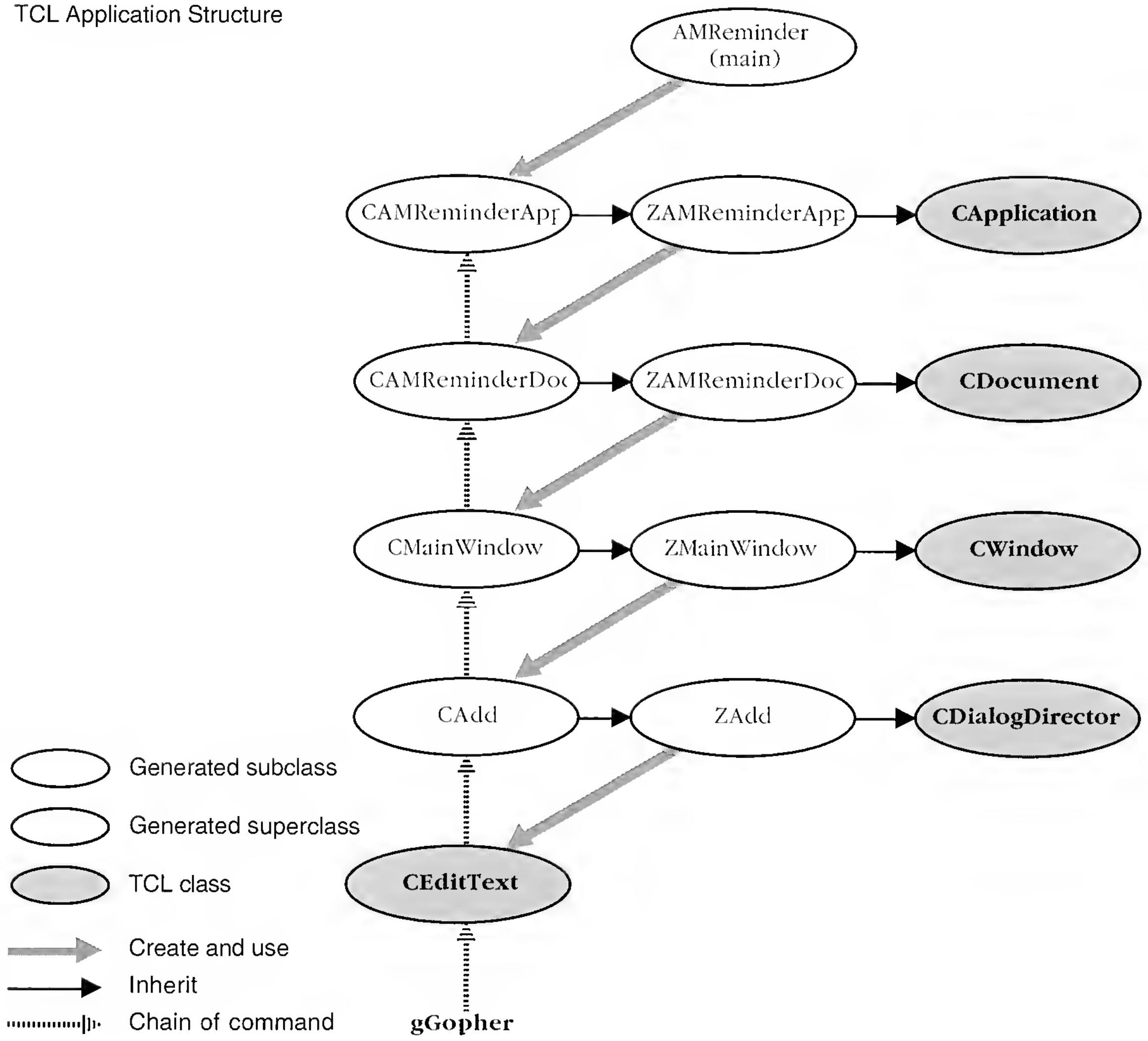
The following sections of this chapter describe the generic structure of the AMReminder application, then provide details of the individual modules.

Differences in Object Program Structure

Object-oriented programs are quite different in structure from procedural programs, as illustrated by the following figure. This figure shows how the completed AMReminder application's modules fit together and how commands flow through the system of classes.

You'll see three types of modules (depicted as ovals in the figure). The white ovals represent subclass modules that have been generated by AppMaker, but which we have significantly modified to implement the details of the AMReminder application. The shaded ovals represent unmodified AppMaker-generated superclasses of the subclass modules. Finally, the dark ovals represent classes in the THINK Class Library (TCL), or in AppMaker's extended class library modules (AMClassLibC).

TCL Application Structure



Object-oriented applications are sometimes described as being *upside down*, when compared to procedural applications. In a procedural application, the code usually makes use of library functions and calls these, as needed, to perform its task.

Object-oriented applications, on the other hand, contain methods that are called by the *class library* being used. The class library (which in our case is the TCL) calls the application's methods when events occur that the library is not equipped to handle. In many cases, the class library is capable of handling the majority of user-generated events, including moving and resizing windows, scrolling windows, handling menu selections, opening and closing desk accessories, and a large number of other chores.

Referring back to the previous figure, you'll see that there are three ovals associated with each major element of the application. The TCL requires that the main functions of the application be performed in a class that is a descendent of its CApplication class (depicted in a dark oval). In our case, AppMaker has generated a class called ZAMReminderApp (depicted in a shaded oval), and a subclass to that called CAMReminder (depicted in a white oval).

AppMaker's code generation methodology, for applications that use the TCL, creates methods that are unlikely to need modification in the superclass (shaded ovals), but you can add override methods in the corresponding subclass (white oval). Methods that are likely to be modified are generated in the subclass. All method calls are first directed to the subclass. If the subclass doesn't contain an override for the method, the inherited method is used. If the superclass doesn't contain an override for the method, the TCL's class (or one of its ancestors) will handle the call.

Thus, calls directed to the CAMReminderApp class may be handled by override methods contained in that class, may be handled by methods in the ZAMReminderApp superclass, or, finally, by the TCL's CApplication class.

The TCL also defines the notion of a document. Document classes are generally associated with a file and one or more windows. So AppMaker generates a CAMReminderDoc subclass and a ZAMReminderDoc superclass, the ancestor of which is the TCL's CDocument class. Calls directed to the CAMReminderDoc class may be handled by override methods in that class, may be handled by methods in the ZAMReminderDoc superclass, or, finally, by the TCL's CDocument class.

For each window defined in the application's interface design AppMaker generates a subclass and a superclass named after the window name. The superclass inherits methods from its ancestor, the TCL's CWindow class.

In the case of the AMReminder application, a single window (called MainWindow) was defined. Therefore, code for the subclass CMainWindow and its superclass ZMainWindow has been generated. All calls to methods are first directed to the CMainWindow subclass, then to the ZMainWindow superclass, then, finally, to the TCL's CWindow class.

The AMReminder application also has a single dialog called the Add dialog. AppMaker generates files for the CAdd subclass and ZAdd superclass. These inherit their behavior from AppMaker's CAMDialogDirector and the TCL's CDialogDirector classes. A global function called DoAdd is generated in the CAdd module to open and initiate operation of the modal dialog.

The Chain of Command

In the TCL, the CSwitchboard class handles the distribution of all events. Mouse clicks are sent directly to the DoClick method in the pane in which the click occurred. Menu commands (or their command-key equivalents) are sent to the DoCommand method of the class whose handle is contained in the **gGopher** variable. If that class (or its ancestors) can't handle the command, it is passed up the list of bureaucrats (kept by the TCL), in a list called the **chain of command**.

The chain of command is dynamic; it changes at runtime as windows and items are activated and deactivated. It generally starts with an active edit text field, then points to its supervisor, which point to its supervisor, and so on up the chain. In the preceding figure, the gGopher variable points to an edit text field in the Add dialog. From there, the chain of command passes upward through the Add dialog, the MainWindow, the document, and finally the application.

Since each object in a TCL application is a subclass of a subclass and so on, each object's DoCommand method generally calls its inherited DoCommand method, which calls its inherited DoCommand and so on, before finally calling its supervisor's DoCommand method.

The following sections of this manual discuss the specifics of the generated and manually-added code in the AMReminder application.

Preliminary Initialization

Execution of the AMReminder application begins in the **main** function, as in any other C language application. This function is located in the AMReminderMain module, which AppMaker has generated, and which requires no modification.

The Main code refers to a global variable called gApplication, which contains a handle to the Application object that is present in every application that uses the TCL. Any method in the application that needs to send a message to the Application can do so by using this variable. AMReminderMain creates a single instance of the class CAMReminderApp and stores it in the gApplication variable.

The main function initializes the Application class by calling the IAMReminderApp method and using the gApplication global variable to reference the class. The IAMReminderApp method is contained in the ZAMReminderApp module, which was generated by AppMaker.

Note that the call to the IAMReminderApp method is directed to the AMReminderApp module, and not the ZAMReminderApp module. This is because the gApplication global variable points to the subclass, not the ZAMReminderApp superclass. Because of the wonderful feature of "inheritance" in object programs, the IAMReminderApp method is available to any instance of the CAMReminderApp class. This simply means that if a method resides in the subclass, or any of its (ancestor) superclasses, it is always accessible to the subclass. In this case, the superclass (ZAMReminderApp) initialization method merely passes the call and its parameters to the CApplication class in the TCL.

Running the Application

When the IApplication method completes execution, control returns to the main function in the AMReminder module. At this point, the application sends another message to the Application class by using the gApplication variable to call its Run method. This method isn't present in the CAMReminderApp or ZAMReminderApp classes, so it propagates up to the TCL's CApplication class to execute.

In System 7, one of the first events an application receives, is an AppleEvent for OpenApp (Open Application). When the TCL receives this event, it calls the application's CreateDocument method. This is an empty method in CApplication; AppMaker generated an override method in ZAMReminderApp.

This method creates an instance of the CAMReminderDoc class, which is a subclass of the ZAMReminderDoc superclass. Following this, the IAMReminderDoc method is called (implemented in the ZAMReminderDoc superclass), which merely calls the TCL's IDocument method.

Once the CAMReminderDoc instance is created, its NewFile method is called. Both the NewFile and BuildWindows methods are in the ZAMReminderDoc module, as generated by AppMaker, and have not been modified. The NewFile method calls the BuildWindows method, then calls the newly-created window's Select method, to make it the active window.

The BuildWindows code is quite simple. It creates an instance of the CMainWindow class (which is a subclass of the ZMainWindow superclass), and initializes the instance by calling its IMainWindow method. IMainWindow, in turn, calls IZMainWindow in its ZMainWindow superclass.

The IZMainWindow initialization method creates instances of all the interface element objects in the main window's enclosure. This includes:

- ☐ The AMReminder CPicture object
- ☐ The CAMStaticText objects
- ☐ The "ScrollPane8" scrolling list pane object, its "Rect7" CBorder object, and its associated "List10" CList10 object
- ☐ The three CButton objects (AddButton, EditButton, and DeleteButton).

In IZMainWindow, the main window is created, with all its fields and controls. When the superclass method has completed execution, the subclass adds to the window's functionality by setting a double-click command for the List10 scrolling list and initializes its associated array. The functionality for the SetDblClickCmd method is provided in the CTable class. It automatically generates the specified command when a double-click occurs in the array pane.

AppMaker generated all of the code in the ZMainWindow module's IZMainWindow method, and also the method and its call of the inherited version of the IMainWindow method in the MainWindow module. Notice that the List10 object is created by a method called MakeList10. Having a separate method for this purpose allows us to override it in the CMainWindow subclass.

The TCL's CTable class, from which the scrolling list ultimately inherits its behavior, requires that the user provide a subclass of that class to explicitly override its GetCellText method. Because of this requirement, AppMaker generates the MakeList10 method in both the ZMainWindow and MainWindow modules.

Because the subclass's MakeList10 method does not call the corresponding inherited method in the superclass, the subclass's method is the only one that executes. It completely overrides the superclass version. The itsData instance variable in the CMainWindow class is saved into an instance variable in the subclass's MakeList10 method, so that the methods in the CAMReminderData class can be referenced by the CList10 subclass methods.

Completing AMReminder's Code

After the new main window has been opened, the application will have reached a stable execution state. At this point, all the initialization has been performed and the application is waiting for the TCL to send events for it to process. In this state, the scrolling list is empty and the Add button is enabled, but the Edit and Delete buttons are disabled. (The Edit and Delete buttons were defined to be initially disabled in the user interface design.)

In the following sections, the modifications to AppMaker's generated code will be discussed, for it is these modifications that provide AMReminder with its unique functionality. Modifications to AppMaker's generated code will be made only to the subclass modules. The following sections describe each addition or modification we made to AppMaker's generated code for the AMReminder application.

Updating Menus

The TCL automatically disables all menu items, then calls the gGopher's UpdateMenus method. Within the TCL and in most applications there are many, many UpdateMenus methods. Each one calls its inherited UpdateMenus, then enables any menu items that it is prepared to handle. The net effect of calling the gGopher's UpdateMenus method, is that all menu items which can be handled by any bureaucrat along the current chain of command, are enabled.

AppMaker generates code to enable all menu items other than those reserved by the TCL. The generated code handles menu items in either the App (Application) class or the Doc (Document) class. Menu items that invoke modeless dialogs are generated in the App class; all other items are generated in the Doc class.

To properly handle setting up menus, then enable or disable the individual items, we need to expand the UpdateMenus method in the CAMReminderApp subclass. We added code to enable the Hide Reminders command if the application has any open documents.

Notice that in the UpdateMenus method, we call the EnableCmd method in the class whose handle is stored in the gBartender global variable. Although there is only one class that handles events associated with the menu bar, its handle is stored in a global variable so that its methods are available to all other classes in the application.

In addition to enabling the Hide Reminders command, we also added code to implement this command. Hide Reminders hides all open reminder windows or shows them if they are already hidden. In CAMReminderApp's DoCommand method we call a new method, ShowOrHide, which we wrote.

ShowOrHide toggles an instance variable isHidden that we added to keep track of whether the reminder window(s) are currently hidden or not. It also toggles the wording of the Hide Reminders command between "Hide Reminders" and Show Reminders." ShowOrHide then loops through all of the application's document (CAMReminderDoc) objects telling each one to show or hide itself.

In the CAMReminderDoc class we wrote another method, also called ShowOrHide, which does the actual showing or hiding of the document's window. Because CAMReminderApp and CAMReminderDoc are separate classes, the scope rules allow the same name to be used in both of them.

Accessing Data

AppMaker generates a separate module, called Data, that defines a subclass of the CDataFile class in the TCL, to handle all of the file-oriented I/O chores. Methods in this class include those needed to handle the commands in the File menu that perform Open, Save, Save As, Close, and Revert operations.

The purpose of the Data module is to isolate all the application-specific data access methods. For almost all applications you will modify it to suit your application. AppMaker generates just a skeleton of this module.

Quite a bit of the generated code can be used as is; however, we need to add the necessary code to fully implement the IAMReminderData, ReadData, and WriteData methods in this class. The CAMReminderData instance is created by the NewFile and OpenFile methods in the ZAMReminderDoc superclass, when the user chooses the New or Open commands, respectively.

In addition to the code generated by AppMaker, the IAMReminderData method must initialize the itsData variable to NULL, create the instance of CArray to be used for holding reminder entries, and initialize the itsArray instance variable.

The new ReadData method is called by the data class's OpenData method, which, in turn, is called by the ZAMReminderDoc class's OpenFile method in response to an Open command being selected from the File menu.

The skeleton for the ReadData code was generated by AppMaker; however, it is oriented toward applications that read the entire contents of the file into a single data handle. For our purposes, it is better to read the data associated with a single reminder at a time.

AMReminder's file format consists of a set of contiguous reminder entries. In order to separate the I/O from the data structure, and also to encapsulate the methods that operate on the data structure, we have created a separate module called ReminderEntry that includes the CReminderEntry class, the reminder structure, and its access methods.

By partitioning the I/O and access methods in this way, the ReadData method is very straightforward. It begins by setting the file to its beginning, gets the length of the file, then enters a loop that reads and processes the data for one reminder at a time, until the end-of-file is reached. Each time through the loop, it executes the ReadSome method inherited from CDataFile to read the data for a single reminder into the aReminder variable. Then it creates a new instance of the CReminderEntry class and stores the data fields from aReminder into the corresponding CReminderEntry instance variables by using the instance's access methods. After the instance has been fully initialized, its handle is added to the end of the array associated with the scroll pane (List10). After that, the length of remaining data is decremented by the size of the reminderData structure.

We have also added code in the CAMReminderData class to handle the WriteData case. This method is invoked when reminders are to be saved to disk, by selection of the Save or Save As commands.

The skeleton outline for the WriteData method was generated by AppMaker; however, we have written all new code for this method. AppMaker often puts "empty" methods into the files it generates, to indicate that these must be enhanced to provide the unique functionality the application requires. This was the case with the WriteData method.

The code for the WriteData method is very straightforward, and mirrors that of the ReadData method. In the case of WriteData, the file is initially positioned at its start, the number of CReminderEntry instances in the array associated with the scrolling list is accessed and a loop iterates through the set of entries. The instance variables in each CReminderEntry are placed into corresponding fields of the aReminder structure and the WriteSome method is used to write them to the file. When the loop is complete, the file's length is set to the position just beyond the last entry.

The Revert method is rather simple. It calls the DisposeData method to remove each CReminderEntry instance from the array associated with the scrolling list by calling a new method named DeleteAMReminder, that we've added to the CAMReminderData class. After deleting all the entries, Revert calls the ReadData method to accomplish the Revert operation.

The DeleteAMReminder method is also used by the CMainWindow class to delete a reminder when the user highlights it in the scrolling list, then clicks the Delete button. It merely calls the array method to delete the specified entry, and also marks the file as dirty.

The remaining methods that we have added to the CAMReminderData class are those to support acquisition of the array instance, getting, setting, and adding reminders to the array (for supporting the Add, Edit, and Delete buttons in the CMainWindow pane). These methods will be described along with the other CMainWindow functions, below.

Main Window Modifications

All of the following additions and modifications pertain to the CMainWindow class and the array support methods in the CAMReminderData class.

The revisions to the IMainWindow method concern the addition of a click command to support a double-click in the scrolling list pane. The array associated with the scrolling list is installed as the last action of the subclass's IMainWindow method. The array instance is accessed by calling the GetArray method in the CAMReminderData class.

The entire purpose of the GetArray method is to return the CArray instance that was created by the IAMReminderData method in the CAMReminderData class.

The DoCommand method for CMainWindow is completely generated by AppMaker. We didn't have to modify it; however, it is instructive to show how the click commands we installed are dispatched.

The DoCommand code contains three cases. These are cmdAddButton, cmdEditButton, and cmdDeleteButton, corresponding to the Add, Edit, and Delete buttons. Each of these cases calls an appropriate method to handle the task of implementing the logic for its associated button.

The DoAddButton code is responsible for creating a new CReminderEntry instance and initializing its fields by using the appropriate access methods.

Once the fields of the CAMReminderEntry have been initialized as described, the DoAdd function is called. At this point the DoAdd function will display the modal dialog and interact with the user. The user will need to type in a date, time, and a message, and may or may not change the state of the check boxes or radio buttons. If the entry is valid and the OK button is clicked, the modal dialog is dismissed and the DoAdd function returns TRUE.

Continuing the discussion of the DoAddButton method, in the CMainWindow class, we find that if the DoAdd function returns a FALSE result, then nothing further is done. If TRUE is returned, then a new entry has been made. In this case, the code inserts the new entry at the end of the list of reminders by calling the AddAMReminder method in the CAMReminderData class.

The Merits of Collaboration

At this point, it is worthwhile to take a short side trip to examine how the TCL handles the addition of an entry to the CArray class instance. If you have examined the structure of the THINK Class Library, you've probably noticed that the majority of the classes inherit behavior from a class called CCollaborator. This class provides the ability for classes to send messages about changes to their objects to other interested classes. This process of notification is called **collaboration**, and is supported by quite a few of the TCL classes. In our case, the CArray class calls the CCollaborator's BroadcastChange method inside its InsertAtIndex method, indicating that the array's contents have changed.

The BroadcastChange method searches its list of dependents and finds that our scrolling list pane *depends upon* changes to the associated array (this is established when the list's SetArray method is called). Therefore, it calls the scrolling list's ProviderChanged method to notify it of the change.

Our CList10 class doesn't include a ProviderChanged method, but it inherits this method from its CArrayPane superclass. This method tests its first argument to determine the nature of the change (arrayInsertElement, arrayDeleteElement, arrayMoveElement, or arrayElementChanged) and calls the appropriate method to handle the change (AddRow, DeleteRow, etc.). Each of these methods eventually calls a method named GetCellText to supply the text for the changed element in the list pane. Our CList10 class overrides the GetCellText method to supply a string that constitutes the abbreviated reminder entry that is displayed in the list pane for that cell.

The GetCellText method is called with a Cell, an available width, and a string pointer. The Cell argument provides us the means to identify the array element to access for acquiring the requested information. The contents of the associated CArray instance are used to create a formatted string that is stored into the area addressed by the string pointer argument.

This is all that is required for the scrolling list to be automatically updated with the new entry. We don't have to write a single line of code to explicitly make entries into the scrolling list. The CArrayPane and its ancestor CTable class will call our GetCellText method whenever the contents of the scrolling list pane need to be redrawn. This is the advantage of the collaboration mechanism.

Back to the MainWindow Methods

When the Edit button in the main window is enabled (which will be the case only when a reminder in the scrolling list is selected), clicking this button will produce a cmdEditButton command, which is handled by the DoCommand code to call the DoEditButton method.

The DoEditButton method must first determine which entry was selected, by calling the GetSelect method for the list pane class. This method returns the row number of the selected entry in our aCell variable. If an entry was selected (which must logically be the case; otherwise, a program error has occurred), then the DoEditButton method proceeds to perform its intended actions; otherwise, no further actions take place.

The selected entry is available as the vertical coordinate of the theCell variable (lists can be two-dimensional, with columns as well as rows). We access the CReminderEntry object for the selected entry by calling the GetAMReminder access method with the entry number. This retrieves the appropriate CReminderEntry instance from the array. If accessing that entry was successful (it should be), the DoAdd function is called (to be described later) and if it returns a TRUE result (indicating that the OK button was clicked), the SetAMReminder access method is called to replace the contents of the entry in the array. The SetAMReminder method also sets the dirty instance variable for the associated data file to TRUE.

When the Delete button in the main window is enabled (which will be the case only when a reminder is selected in the scrolling list pane), clicking this button will cause a cmdDeleteButton command to be intercepted by the main window's DoCommand method. This will result in a call to the DoDeleteButton method to handle this command.

The DoDeleteButton method, like the DoEditButton method, applies only to a selected entry in the main window's scrolling list pane. Its first action is to ascertain which entry was selected, by calling the GetSelect method for the scrolling list. This method places the selected cell's coordinates into our aCell variable, and returns TRUE if a selected entry was found. If not (which should never occur), the DoDeleteButton will simply return, without performing any further actions. If an entry has been selected, its entry number is taken from the vertical coordinate of our aCell variable and the CReminderEntry instance associated with this entry is retrieved by calling the GetAMReminder access method.

By virtue of the collaboration mechanism described above, changes to the array are reflected in the entries displayed in the scrolling list pane. When an entry is deleted, all the entries below it (if any) are moved up one row and the scrolling list pane is refreshed. All of this happens automatically, behind the scenes.

One additional feature of the collaboration mechanism is of interest at this point. Quite a number of the TCL classes inherit behavior from the CBureaucrat class. This includes the CApplication, CDocument, and all the CView classes and subclasses. The CBureaucrat class overrides the BroadcastChange method to inform the supervisor of the class that called that method of the change, as well as calling the inherited BroadcastChange method. This provides notification of certain changes to the document or window classes containing elements that call the BroadcastChange method.

When a mouse click occurs within the scrolling list pane, the SelectRect method of the CTable class (from which our CList10 class inherits behavior) highlights the selected cell and calls the BroadcastChange method. This is intercepted by CBureaucrat which calls the ProviderChanged method (if any) for the supervisor of the scrolling list (in this case, our CMainWindow class).

The purpose of the ProviderChanged override method in CMainWindow is to enable or disable the Edit and Delete buttons, depending on whether an entry in the scrolling list is highlighted or not, respectively.

AppMaker generated the code to test whether anything was selected in the list. We added the code to activate or deactivate the Edit and Delete buttons.

Add Module Modifications

When the user clicks the Add or Edit buttons in the main window, or if an entry in the scrolling list of reminders is double-clicked, the Add module's DoAdd function is called. This function is not a method, but a global function in AppMaker's generated code. It is accessible from any other class in the application.

The DoAdd function creates a new instance of the CAdd class, then calls the IAdd initialization method contained in the ZAdd module.

The initialization code was entirely generated by AppMaker and we do not have to override or change it in any way. Basically, this code creates instances of each of the CAdd dialog's user interface objects and initializes them. Once this is done, then the DoAdd code can set the proper contents of each of the elements, according to the corresponding fields in the CReminderEntry argument passed to the function. Once again, the methods associated with the CReminderEntry instance are used to access its contents.

After the fields of the CReminderEntry instance have been stored into the corresponding elements of the dialog, the DoAdd function calls the dialog's DoModalDialog method, which it inherits from the CDialogDirector class. This method takes care of all the interaction with the user and returns only when either the OK or Cancel buttons is clicked. If the Cancel button is used to dismiss the dialog, then the dialog is disposed and the function returns a FALSE result. If the OK button was used to dismiss the dialog, the content of the dialog's elements are stored back into the CReminderEntry instance, using its access methods. When the instance has been updated, the dialog is disposed and the function returns a TRUE result.

The CAdd class also overrides the ProviderChanged method to accept BroadcastChange messages from the dialog pane's providers. In our case, this method is interested in determining whether text has been entered into the DateField, TimeField, and MessageField panes.

The ProviderChanged override method illustrates how the collaboration mechanism works for subpanes of a dialog window. The BroadcastChange messages that we're interested in are generated by the SelectionChanged method in the TCL's CAbstractText class, from which our CAMDialogText panes inherit their behavior. Some of the other CAdd dialog elements also call the

BroadcastChange method, but we are not interested in their changes. The ProviderChanged method identifies the providers in which it is interested by the first argument to the method call. In our case, the only providers in which we are interested are the DateField, TimeField, and MessageField dialog items. Our intention in observing changes to these fields is to ensure that each contains an entry, and to enable or disable the OK button in the dialog if all three of the fields haven't been filled (a more rigorous validation technique would be needed in a real application; however we have chosen not to burden you with this detail).

AppMaker generated code to test whether the three text fields had any data. We manually added code to activate or deactivate the OK button.

CReminderEntry Module Methods

In order to hide the details of the structure of reminders and provide methods to access the various pieces of information that make up a reminder, we have created a new class called CReminderEntry. This class defines an object that encapsulates the data fields and methods for handling reminder entries.

We have arbitrarily defined CReminderEntry as a subclass of CCollaborator. This has no significance, and the class could as easily have been made a subclass of any other class in the TCL hierarchy, including CObject. At the outset, it was thought that there might be some need to issue BroadcastChange messages, although that hasn't turned out to be the case. If a more robust error checking methodology was employed, then such a mechanism might be warranted.

The CReminderEntry class has a lot of very simple methods. They simply copy data from a parameter to an instance variable or return the value of an instance variable. This technique is used very frequently in object-oriented programming to isolate implementation details from other modules.

Creating a separate class and access methods for the reminder entries simplifies the overall code and makes it more readable.

What's Missing in the AMReminder Application?

The classes and methods discussed in the previous sections of this chapter have covered all the pertinent routines that implement the specifics of the AMReminder application's user interface. However, none of these routines has contributed to the implementation of using the reminder entries to notify the user when it is time to do so.

The missing code in the AMReminder application must accomplish the following tasks:

- ☐ Using a Dawdle method to call, in turn, each Bureaucrat in the chain of command, commencing with the module associated with the gGopher global variable (which, in our case would be the CMainWindow), the application should periodically examine the list of reminders and determine if any require the posting of a notification.
- ☐ Provide a new method to format a request to the Macintosh Notification Manager and call the Notification Manager's NMInstall toolbox routine to post the notification.
- ☐ When a notification is posted, the Cancel Reminder command in the Remind menu should be enabled.
- ☐ Provide a new method to respond to the Cancel Reminder command, in the Remind menu, which calls the Notification Manager's NMRemove toolbox routine to clear the notification and deactivate the Cancel Reminder command.

In the TCL version of our AMReminder application, about 95% of the code is devoted to managing the user interface, and 5% to implement the notification functions. Of the 95%, at least 90% has automatically been generated by AppMaker, or provided in the THINK Class Library. We had to add the remaining 10%; however, this is a small percentage of the overall code.

The remaining 5% would be relatively simple to integrate into the overall framework of the application. The CAMReminderDoc class is the proper place to put most of the code. The UpdateMenus method will need to be enhanced to enable and disable the Cancel Reminder command, at the proper time. This could be based on the addition of an instance variable to the CAMReminderDoc class, that keeps track of notifications that have been issued.

The Dawdle method, for checking reminders, should be added to the CAMReminderDoc class, and the maxSleep value for the application should be adjusted to set a maximum sleep time (when it is running in the background) of less than 60 seconds, so that pending notifications are always posted at their specified times.

Formatting and issuing notifications can be accomplished in a separate routine, called by the Dawdle method. Removing notifications should be handled in the DoCommand method of the CAMReminderDoc class.

AppMaker Library for TCL Applications

The AppMaker product contains a number of library routines, mostly in the form of classes and methods, that pertain to THINK C applications that use the THINK Class Library (TCL). The library routines for THINK Pascal applications, that also use the TCL, are very similar.

The following sections briefly describe the AppMaker library classes. Since almost all of these classes are referenced only by generated code and not by your own code, we have omitted most of the details. The library source code serves as the complete reference.

CAMArrayPane Class

Initializes a CArrayPane object from a pane resource. The pane resource can specify font, size, and style.

CAMButton Class

Initializes a button item from a pane resource. The pane resource can specify font, size, and style.

CAMCheckBox Class

Initializes a check box item from a pane resource. The pane resource can specify font, size, and style.

CAMDialogDirector Class

Initializes a new dialog from a 'DLOG' resource. Overrides the DoCommand method to hide the dialog (rather than dispose it) when a cmdClose command is issued.

CAMDialogText Class

Initializes a CDialogText object from a resource template. The pane resource can specify font, size, style, and justification.

CAMEditText Class

Initializes a CEditText object from a pane resource. The pane resource can specify font, size, style, and justification.

CAMIntegerText Class

Initializes a CIntegerText object from a pane resource. The pane resource can specify font, size, style, and justification.

CAMPopupMenu Class

Initializes a CPopupMenu object and saves a font and size in instance variables. Overrides PopupSelect to set the pop-up menu's font and size.

CAMPopupPane Class

Initializes a CStdPopupPane object from a pane resource. The pane resource can specify font and size. Overrides NewMenuSelection to send a BroadcastChange message whenever the user chooses an item from the pop-up menu. Also includes a PlaySound method to play the sound named by a selected menu item in a Sound menu.

CAMRadioControl Class

Initializes a radio button item from a pane resource. The pane resource can specify font, size, and style.

CAMStaticText Class

Initializes a static text object from a pane resource. The pane resource can specify font, size, style, and justification. It can also specify the resource ID of a TEXT resource which contains the initial text for the object.

CAMStyleText Class

Initializes a CStyleText object from a pane resource. The pane resource can specify font, size, style, and justification.

CAMTable Class

Initializes a CTable object from a pane resource. The pane resource can specify font, size, and style.

CGrayLine Class

Implements a pane containing a gray line object.

CLabeledGroup Class

A subclass of CRadioGroupPane which implements a labeled group of radio button objects. It provides for the group to be enclosed by a rectangular border, and includes an optional label string.

CMultiState Class

Implements the behavior of a multi-picture button. Contains methods to respond to a *good click* and change the state of the button.

CSlider Class

Implements a slider control (like a scrollbar) and provides an override method to respond to the thumb being dragged.

AMUtilities module

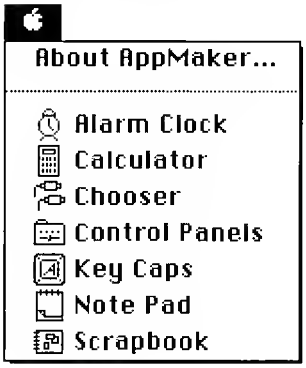
Contains a number of global functions that are not class methods, but are called by both the generated code and by methods in AppMaker's class library. In general, these are utility routines for handling the setup of text fonts, sizes, and styles. In addition, it contains the routine to handle initialization of control pane objects (subclasses of class CButton). AppMaker uses pane resources rather than CNTL resources to initialize control objects, because they provide more information. AppMaker's pane resources also specify font, size, and style information for controls and for other objects.

Appendix A
AppMaker's Menus

This appendix briefly describes each of AppMaker's menu commands, in the order from left to right across the menu bar.

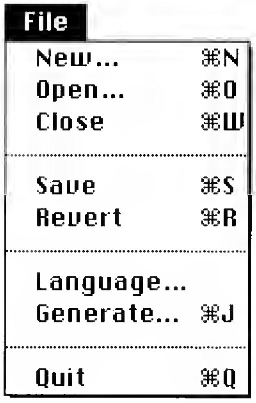
Many menu commands have keyboard equivalents, shown at the right side of the menu. To use the keyboard command, hold down the Command key and type the character shown.

Apple Menu



About AppMaker...
Displays the AppMaker version number and copyright notice.

File Menu



New...
Creates and opens a new AppMaker document.

Open...
Opens a file (application, AppMaker document, ResEdit document, or ViewEdit document) that has been created previously.

Close
Closes the currently active window. Closing the List window also closes the Menu window or Items window and closes the current file.

Save
Writes to the file any changes you have made since you last opened or saved the file.

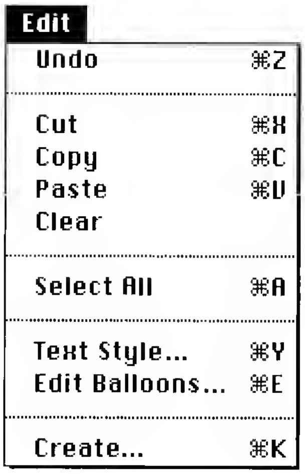
Revert
Has the same effect as closing and not saving changes, then reopening the file.

Language...
Lets you select a language system to be used for creating resources and generating code for the user interface.

Generate...
Generates source code files in the selected language.

Quit
Closes any open files, saves changes, exits AppMaker, and returns to the Finder.

Edit Menu



Undo
Undoes the most recent change (typing, Cut, Copy, Paste, Clear, moving, or reshaping).

Cut
Removes the selected resource, item, or text and puts it on the Clipboard.

Copy
Copies the selected resource, item, or text to the Clipboard.

Paste/Paste Picture/Paste Window/etc.
Pastes a resource, item, or text from the Clipboard.

Clear
Deletes the currently selected resource, item, or text without copying it to the Clipboard.

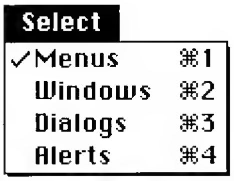
Select All
When editing text, selects all the text in the current field. When an Items window is active, selects all the items in the Items window.

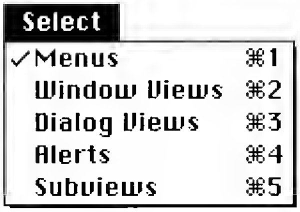
Text Style...
Displays a dialog for changing the font, size, style, and justification of the selected item(s) in an Items window.

Edit Balloons...
Displays a modeless dialog of Help balloons for the selected item in an Items window, so you can create and edit Balloon Help for up to four states of an item: enabled, disabled, checked, or marked.

Create Menu Bar.../
Create Window.../Create Dialog.../Create Alert.../
Create Menu/Create Submenu...
When the List window is active, creates a new menu bar/window/dialog/alert. When the Menu window is active, creates a new menu. When the command-key equivalent area of a menu item is active, creates a new submenu.

Select Menu





The Select menu provides access to all the resources you can create, change, or delete using AppMaker's User Interface Editor.

The Select menu contains a mutually-exclusive group—only one command is checked at a time.

Menus
Displays in the List window all the menu bar resources already defined in the application. Click menu bars in the List window to successively display the Menu window for each menu bar, or click in the Menu window to begin editing menus in the selected menu bar.

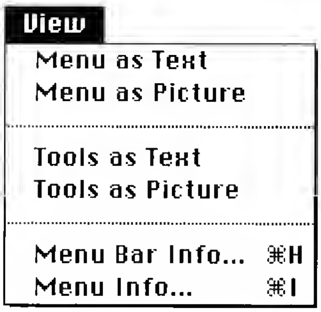
Windows/Window Views
Displays in the List window all the window resources already defined in the application. Click windows in the List window to successively display the Items window for each window, or click in the Items window to begin editing items. (For MacApp, this command is called Window Views.)

Dialogs/Dialog Views
Displays in the List window all the dialog resources already defined in the application. Click dialogs in the List window to successively display the Items window for each dialog, or click in the Items window to begin editing items. (For MacApp, this command is called Dialog Views.)

Alerts
Displays in the List window all the alert resources already defined in the application. Click alerts in the List window to successively display the Items window for each alert, or click in the Items window to begin editing items.

Subviews
For MacApp only, displays in the List window all the subview resources already defined in the application. Click subviews in the List window to successively display the Items window for each subview, or click in the Items window to begin editing items.

View Menu



Menu as Text
Displays the selected menu as text (a standard Macintosh menu).

Menu as Picture
Displays the selected menu as a picture. If the menu has no picture, displays a dialog box asking you to select a picture.

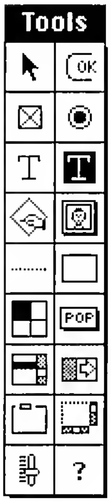
Tools as Text
Displays AppMaker's Tools menu as a textual list of tools in alphabetical order, instead of a palette of pictures of the tools. Additional tools are accessible for the THINK Class Library and for MacApp when you view the Tools menu as text.

Tools as Picture
Displays AppMaker's Tools menu as pictures of the tools, described under the Tools Menu. Some tools that are available for the THINK Class Library and for MacApp aren't shown when you view the Tools menu as pictures. See "Tools as Text" above.

Menu Bar Info.../Window Info.../Dialog Info.../Alert Info...
Displays and lets you change information about the currently selected menu bar/window/dialog/alert.

Menu Info.../Item Info...
Displays and lets you change information about the currently selected menu (if a menu is selected) or item (if the Items window is active).

Tools Menu



The Tools menu is a palette menu—only one tool is selected at a time. The currently selected tool is displayed with a thick border. Tools are used to create items in windows, dialogs, and alerts.

You can also view the Tools menu as text (by choosing Tools as Text from the View menu), which gives you access to additional tools for the THINK Class Library and MacApp.

Command-\ is a keyboard shortcut that alternates between the currently selected tool and the Arrow tool.

Select items with the **Arrow** tool (by clicking, dragging, or Shift-clicking), so you can change them in various ways, such as moving, reshaping, aligning, or deleting.

Click the **Button** tool to create a push button at the pointer location. AppMaker displays an insertion point inside the button so you can type the button's title.

Click the **Check Box** tool to create a check box at the pointer location. AppMaker displays an insertion point to its right so you can type a label for the check box.

Click the **Radio Button** tool to create a radio button at the pointer location. AppMaker displays an insertion point to its right so you can type a label for the radio button. To be considered a radio button group in procedural languages, the radio buttons must have consecutive item numbers; in the THINK Class Library and MacApp, they must be enclosed in a labeled group or cluster, respectively. For the TCL, radio buttons can also be enclosed in a CRadioGroupPane (which can be created when you view the Tools menu as text).

Click the **Static Text** tool to position the insertion point for typing static text. Press the Return key to force text to start a new line. Or, reshape the static text item after you've created it to adjust the width of the text. AppMaker will automatically re-wordwrap the text.

Drag with the **Edit Text** tool to create a field in which the user will type text. When you release the mouse button, AppMaker displays an insertion point so you can type a title that is used in the generated source code. You can reshape the text to control the width and number of lines. For procedural languages, you can create a numeric field by typing a number sign (#) at the end of the field's title.

Click the **Icon** tool where you want the upper-left corner of an icon to be. AppMaker displays a dialog box that lets you choose the icon from the ICON resources available in your document or System file. To create an icon button (available only for procedural languages), hold down the Option key while clicking with the Icon tool. (Icon buttons act like radio buttons instead of being static images.)

Click the **Picture** tool where you want the upper-left corner of a picture to be. AppMaker displays a dialog box that lets you choose the icon from the PICT resources available in your document. To create a picture control (a picture that responds to mouse clicks), use the Custom Control tool instead of the Picture tool.

Drag with the **Line** tool to create a horizontal or vertical gray line. You can also create a diagonal line provided it slopes downwards to the right.

Drag with the **Rectangle** tool to create a rectangle to visually group items. Items within a rectangle are nested; any part of the enclosed item that extends beyond the bounds of the rectangle is not shown. To define radio buttons in the THINK Class Library of MacApp, use the Labeled Group tool instead of the Rectangle tool.

Click the **Palette** tool to display a dialog that lets you choose the PICT resource in your document to use for the palette, tell AppMaker the number of items across and down in the palette, and choose how a selected item is displayed (by specifying a frame width).

Click with the **Pop-up** tool where you want the upper-left corner of a pop-up menu to be. AppMaker displays a dialog that lets you specify the menu to use for the pop-up item (either an existing or new submenu). You use AppMaker's menu editor to type the menu items in the submenu.

Drag with the **List** tool to create a single-column list with a vertical scroll bar. The contents of the list at run-time are defined by your code. (For the THINK Class Library and MacApp, there are additional steps for creating a list as described in Chapter 3, "Creating and Editing the User Interface" in the section on the List tool.)

Drag with the **Scroll Bar** tool to create a scroll bar. For procedural languages, if you create a scroll bar that touches the right or bottom edge of a window, AppMaker will extend the scroll bar to the window's grow icon. (To create a scrollable region in the THINK Class Library or MacApp, use the Scroller tool instead of the scroll bar tool.)

Drag with the **Labeled Group** tool to create a grouping rectangle for items. After you've dragged the labeled group to size, AppMaker displays an insertion point for you to type the label. If you don't type a label, the labeled group itself will be invisible, but will still provide a container for the items within it. For example, radio buttons in the THINK Class Library or MacApp must be enclosed in a labeled group in order to be recognized as radio buttons.

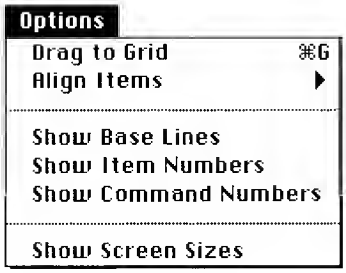
Drag with the **Scroller/ScrollPane** tool to create a scrollable region for the THINK Class Library or MacApp. (Scrollers are not available for procedural languages.) A scroller is initially created with a vertical scroll bar, but you can add a horizontal scroll bar using the Scroll Bar tool, or delete scroll bars to leave just the scroll region.

Click with the **Custom Control** tool to display a dialog that lets you place a control for which you've created your own CNTL resource, or to create a built-in custom control. AppMaker's built-in custom controls include: picture button, picture check box, picture radio button, multi-picture button, and slider. To create the control, you choose the PICT resources in your document that show the states of the control, and specify some other information about the behavior of the control.

?

Drag with the **User Item** tool to create an item defined by your code. AppMaker displays a user item as a rectangle with a thick gray border.

Options Menu



Drag to Grid
Helps you line up items in the Items window. When Drag to Grid is on, there is an invisible grid every four (4) pixels. When you drag an item, its top left corner is aligned to the nearest grid intersection. When you resize an item, its size is made a multiple of the grid size. When you resize an Items window, the window size is made a multiple of the grid size. Drag to Grid is normally on.

Align Items
Aligns the selected items by their top, left, bottom, or right edges, or by their centers horizontally or vertically.

Show Base Lines
Turns on and off the display of base lines for multi-line items (such as static text or edit text). Show Base Lines is normally on.

Show Item Numbers
Turns on and off the display of item numbers. Show Item Numbers is normally off.

Show Command Numbers
Turns on and off the display of command numbers for menu items. (Relevant for THINK Class Library and MacApp languages only.) Show Command Numbers is normally off for Procedural languages, and normally on for TCL and MacApp languages.

Show Screen Sizes
Turns on and off the display of a background window that shows the screen sizes of the standard 9" and 13" Macintosh screens.

Appendix B
What's New in AppMaker 1.5

This appendix lists the features that are new in Version 1.5 from the previous Version 1.1 (for Procedural and THINK Class Library) and Version 1.2 (for MacApp).

AppMaker 1.5:

- Is System 7.0-friendly
- Takes full advantage of TCL Version 1.1
- Supports font, size, style
- Supports the latest versions of all languages
- Includes many other new features
- Has improved documentation, including a new example application

The remainder of this appendix is divided into sections for the AppMaker application and for the different language categories: Procedural, TCL, and MacApp. There is also a section on compatibility between AppMaker 1.5 and previous versions.

Changes to the AppMaker Application

- ☐ AppleEvent–aware
- ☐ Stationery–aware; uses stationery documents
- ☐ Supports Movable Modal dialogs
- ☐ Has a Balloon Help editor
- ☐ Has Balloon Help, including Help for its picture (Tools) menu
- ☐ Can generate code while running in the background
- ☐ Has new Finder icons in all colors and sizes
- ☐ Supports all of the new TCL 1.1 classes
- ☐ Window and dialog items can be nested
- ☐ Text Styles dialog for specifying font, size, style, and justification
- ☐ Show Item Numbers
- ☐ Show Screen Sizes
- ☐ Align Items
- ☐ Move menus in a menu bar by dragging
- ☐ Move menu items within a menu by dragging
- ☐ Picture MDEF supports balloon help
- ☐ Displays Tools menu either as text or as picture
- ☐ New command-key equivalents

Changes for Procedural Languages

- ☐ AppleEvent–aware
- ☐ Stationery–aware
- ☐ Supports Movable Modal dialogs
- ☐ Updates windows behind a modal dialog or alert
- ☐ 'ictb' editor for font, size, and style in resource
- ☐ Item coordinates obtained from resource, not hard-coded
- ☐ Makes color windows if Color QuickDraw is present
- ☐ Main module is generated rather than in library
- ☐ Generates new Data module for isolating application-specific file access code
- ☐ Generates placeholder functions for buttons in windows
- ☐ Generates a DoIdle routine for handling Idle events
- ☐ Generates call to AddResMenu for Sound menu
- ☐ Library support for spinning cursors
- ☐ Improved library support for scrollable lists
- ☐ New library routine to play a sound
- ☐ Utilities for managing linked list
- ☐ Obsolete $load removed from MPW Pascal code

Changes for THINK Class Library Languages

- ☐ Supports all of the new TCL 1.1 classes
- ☐ Creates pane resources for all items
- ☐ Allows nested panes
- ☐ Sets and reads the pane sizing attribute
- ☐ Stores font, size, style in pane resources
- ☐ Lets you specify a class name for any item
- ☐ Supports floating windows and dialogs
- ☐ Generates new Data module for isolating application-specific file access code
- ☐ Modal dialogs are called from Doc class instead of from App class
- ☐ Converts old-style DITL resources to pane resources

Changes for MacApp Languages

- ☐ Supports MacApp 3.0B3
- ☐ Supports font, size, style
- ☐ Supports p1 Modula-2 with MacApp 2.0
- ☐ Edits CMNU, cmnu, and MENU/mntb resources

Compatibility with Previous Versions

We have tried to make Version 1.5 upward-compatible with previous versions. AppMaker itself is a measure of compatibility, because AppMaker is written with AppMaker. We did not have to make modifications to old AppMaker code to run with new library code or new generated code. We made modifications only as we needed to take advantage of new features.

For the TCL, AppMaker takes full advantage of the many improvements since version 1.0 of the TCL. The new TCL does many things very differently from (and much better than) the old TCL. This will cause some incompatibilities between TCL code generated by AppMaker 1.5 and generated by AppMaker 1.1.

Because AppMaker generates two separate files (a superclass and a subclass) for most of a TCL application, we believe that conversion difficulties might be minor. In many cases, you can simply regenerate the superclass with new code, and your overrides in the subclass will still work.

AppMaker 1.5 has built-in automatic conversion from old-style DITL/Witl resources to the new pane resources that AppMaker 1.5 uses. You should be able to just open old documents created by AppMaker 1.1 or by another resource editor and AppMaker will convert them to the new format.

Appendix C
Standard Resources

Whenyou create a new AppMaker document, AppMaker copies many standard resources into the new document. These include Apple, File, and Edit menus, a main window, and alerts such as Save Changes, Revert, and About.

As you design a user interface, AppMaker creates more resources to describe the windows, menus, and other interface elements that you create in the document.

Many of these resources are defined by templates in AppMaker. You can customize AppMaker to change the way it creates resources by modifying these templates. For some resources, you can use AppMaker to modify its templates. For other resources, you can use ResEdit or another resource editor to customize the templates.

This appendix lists the standard resources created for a new application, and describes how you can customize AppMaker to change the created resources.

Stationery Documents

Procedural TCL

MacApp 2.0 MacApp 3.0

AppMaker 1.5 uses stationery documents to define the set of resources it creates in a new document. There is a separate stationery document for each language category: Procedural, TCL, MacApp 2.0, and MacApp 3.0. AppMaker copies all of the resources in the stationery document into a new document. By editing the stationery document, you control what resources are created in a new document.

To edit the stationery document itself instead of creating a new document, you must first change it to an ordinary (non-stationery) document. (If you try to open a stationery document with AppMaker under System 7, AppMaker will create a new document. That's the way stationery documents are designed to work.

1. In the Finder, select the stationery document you want to edit

2. Choose the Get Info command

3. Click the Stationery Pad check box to change the document

With some other resource editors (notably ResEdit as of this writing), you can open a stationery document directly without first changing it to a non-stationery document.

The following sections list the resources found in each stationery document. Many of the resources are essentially the same for the various language categories, and are listed together. They are followed by lists of the language-specific resources.

Stationery Resources for All Languages

- ☐ A Main Menu bar with Apple, File, and Edit menus
- ☐ Balloon Help text for the standard menus
- ☐ A Main Window
- ☐ Alerts such as Save Changes, Revert, and About
- ☐ BNDL, FREF, ICN#, and signature resources for Finder icons
- ☐ icl8, icl4, ics8, ics4, and ics# resources for color and small Finder icons
- ☐ A SIZE resource to specify characteristics of the application
- ☐ A vers resource to specify version information for the Finder
- ☐ A second vers resource to specify the AppMaker version of the stationery document itself
- ☐ An AMKR resource in which AppMaker stored a language selection

Stationery Resources for Procedural Languages

- ☐ Other Alerts which will be needed in the built application
- ☐ ErMs resources which contain the text of error messages
- ☐ A STR resource with the text of a prompt for the Save As dialog
- ☐ A Witl resource to define the items in the Main Window

Stationery Resources for TCL

- ☐ Other Alerts which will be needed in the built application
- ☐ A CNTL resource for a scroll pane's scroll bar
- ☐ Estr resources which contain the text of error messages
- ☐ A Pan# resource to define the items in the Main Window
- ☐ A SICN resource for drawing a standard "size box"
- ☐ A STR resource with the text of a prompt for the SaveAs dialog

- ☐ STR resources with the text of error messages
- ☐ Several STR# resources

Stationery Resources for MacApp 2.0

- ☐ Other Alerts which will be needed in the built application
- ☐ Various Dialog resources
- ☐ errs resources for handling error conditions
- ☐ mem!, res!, and seg! resources for memory management
- ☐ A SICN resource for the MacApp debugger
- ☐ Several STR# resources
- ☐ Several view resources in addition to the MainWindow view

Stationery Resources for MacApp 3.0

- ☐ Other Alerts which will be needed in the built application
- ☐ Various Dialog resources
- ☐ acur and CURS resources for spinning cursors
- ☐ aedt (AppleEvent Dispatch Table) resources
- ☐ A CDEF for popup menu view items
- ☐ errs resources for handling error conditions
- ☐ mem!, res!, and seg! resources for memory management
- ☐ A SICN resource for the MacApp debugger
- ☐ Several STR# resources
- ☐ TxSt resources for specifying text style in View resources
- ☐ Several view resources in addition to the MainWindow view
- ☐ A View resource for the Clipboard window
- ☐ A WDEF for floating windows

AppMaker Templates

In addition to the resources copied from a stationery document when you create a new document, AppMaker creates other resources as you design your user interface. Many of these resources are defined by template resources within AppMaker.

Templates for Windows, Dialogs, and Alerts

When you create a new window, dialog, or alert, AppMaker copies a template resource from itself into your document. These resources define the position, size, window kind, and other attributes of the new window, dialog, or alert.

By editing these template resources, you control how resources are created in your document. For instance, if you want all of your created alerts to have a standard position and size, you can modify the alert template within AppMaker.

These template resources also control the items created automatically in a new window, dialog, or alert. The standard template for a dialog provides OK and Cancel buttons. The standard template for an alert provides a static text item, and an OK button. You can modify the template resources to provide a different set of items.

The best way to edit the window templates is by using AppMaker to edit a copy of itself. You can also use other resource editors. When you select windows, dialogs, or alerts, AppMaker will display template resources along with the window, dialog, or alert resources that are part of AppMaker itself. The template resources are identified by names which begin with "Std. New" (Std. is an abbreviation for Standard).

Templates for TCL Items

For items you create in a TCL window or dialog, there are additional template resources which define those items. The template resource for a CButton defines its initial size to be 20 x 72 pixels and its text style to be Chicago 12. The template for a CTable defines it as allowing only a single selection and not having row or column borders. You can change these template resources so that newly created items have the attributes you want.

This ability to redefine the template resources for TCL items is particularly useful when you want to use subclasses of the standard TCL items. If, for instance, you define a CValidatedText subclass of CEditText, and if you define its resource to have additional fields to specify the validation criteria, you can customize AppMaker to create resources with those additional fields. You can, at the same time, redefine the resource type of an item. Instead of AETx for an Edit Text item, you can have AVTx for a Validated Text item.

You can edit the TCL item templates with ResEdit or another resource editor. The Pan# resource type specifies the resource type and ID of a pane resource for each possible item. The Pan# resource for a CEditText, for instance, specifies pane resource type 'AETx', ID 128. You can edit that resource to change the item's initial attributes, or you can change the Pan# resource to specify a 'AVTx' resource which you define.

Appendix D
Customizing AppMaker's Code Generator

AppMaker is an extremely versatile tool, even if you consider only the spectrum of languages and various class libraries it supports. It is also extremely extensible, which gives you the ability to modify the code it generates to a remarkable degree.

All of AppMaker's generated code is defined by templates stored in the AppMaker application. All are subject to modification, to customize the standard user interface, or to generate code for a completely new language.

You can customize AppMaker to generate code to your own personal or corporate standards, to add features, or to add entire new languages.

AppMaker's code generation is controlled by a Template Language that is implemented as a series of resources stored inside the AppMaker application. Each language (and class library variant) has its own set of templates. When AppMaker begins generating code for a user interface that you've defined, it relies completely on the directives stored in the appropriate set of templates to produce a unique set of modules and application code for that definition.

This chapter provides a specification for the code generation Template Language. It also provides an overview of how the templates are used to generate some of the code that we've described in Chapters 5-6.

AppMaker's Code Generator

AppMaker generates code for your user interface design by interpreting a specific set of templates for the language you've chosen. Each language has a 'Lang' resource which specifies the templates to use to generate code for that language. Each template type is stored inside the AppMaker application as a series of individual resources of the indicated type. The following table lists the template types for each of the various languages.

Language Environment	**Template Type**
Procedural C	TmpC
Procedural Pascal	TmpP
C with THINK Class Library	TmCT
Pascal with THINK Class Library	TmPT
C++ with MacApp 2.0	TmCM
Pascal with MacApp 2.0	TmPM
C++ with MacApp 3.0	TmC3

Note that both THINK C and MPW C, as well as THINK Pascal and MPW Pascal share many of the same template types. AppMaker is able to generate code with the necessary structural and/or content variations needed for programs written in these language versions.

The template types are Macintosh resource types, within which an extensive set of resources are contained. Each of the template resources is a plain text file, containing code written in the specified language, as well as Template Language directives. The directives inside these template resources provide control over the code that is generated for a specific application. The directives are very much driven by the user interface elements that you define, so the resulting application is quite specific to the unique interface for which code is being generated.

The contents of some of the code generation templates will be discussed shortly; however, before delving into those details we will first describe the elements of the Template Language.

Elements of the Template Language

AppMaker's Template Language consists of directives enclosed within percent sign (%) delimiters. The %% combination is used to indicate a literal percent sign is to be generated.

Directives are not case-sensitive; however, they are blank-sensitive. For example, if a directive is stated as: `%for each...%`, then there must be one and only one space between the word "for" and the word "each".

Return Characters

A Return character immediately after the last character (the %) of a directive is not copied to the generated code file. If a Return character is desired, place one or more space characters after the end of the directive, followed by the Return character. In that case, the Return character *will* be generated.

Template Comments

A directive that begins with `%--` (a percent sign and two hyphens) is interpreted as a template comment. It is neither interpreted nor generated.

Tabs

A directive that begins with a percent sign followed by a Tab character generates a variable number of Tab characters. It is used to make generated code line up nicely when there are variable-width items preceding the Tab directive on the line.

Variables

AppMaker's Template Language supports three types of variables. Each is entered in a template in the same way, and each consists of one or two alphabetic words, which are not case-sensitive.

String Variables

When a string substitution variable appears in an AppMaker template, the string associated with its definition is substituted for its name in the generated code. String substitution variables may appear alone in the template (surrounded by percent characters), or they may be part of a more complex statement. In either case, the value of the string is substituted for its name when the directive is interpreted. The following table shows the list of string substitution variables:

Variable	**Value**
appName	name of the application
appFileName	name of the application's resource file
className	class name of the item
date	the date the file is generated
dlogname	name of the dialog
enabledName	name of field to enable the item
enclosure	name of the enclosing item
fieldname	name of the item's value field
filename	name of the generated source file
firstRadio	name of the group's first radio button
handleName	name of the item's handle field
itemName	name of the item
labelName	name of the item's label
lang	which language
menuItemName	name of the menu item
menuName	name of the menu
paneType	resource type of the item's pane
rsrcName	name of the resource

Variable	Value
rsrcType	type of the resource
superClass	class name of the item's ancestor
time	the time the file is generated
unitName	the generated file's unit name
windname	name of the window

Number Variables

Several variables can be used both as string substitution variables and as numeric values which can be tested in an `%if…%` directive

Variable	Value
clickCmd	command number of the item's clickCmd
commandNr	the menu item's command number
iconID	ID of the item's ICON resource
itemNr	the item's number (from 1)
menuID	ID of the MENU resource
menuitemNr	the menu item's item number
paneID	ID of the item's pane resource
pictID	ID of the item's PICT resource
popupID	ID of the pop-up menu resource
procID	procID of the window or dialog
rsrcID	ID of the resource
textID	ID of the item's TEXT resource
viewID	ID of the item's 'view' resource

Boolean Variables

AppMaker's boolean variables are used in conditional code generation directives (covered later). Like the string variables shown in the preceding table, the boolean variables are not case-sensitive. However, you'll notice that some of the boolean variables consist of two words. There must be one and only one space separating the two words in the text of the directive containing these variables.

The following table shows a list of the available boolean variables.

Variable	Boolean value
About	menu item is the "About..." item
Cancel	item is item 2 in a modal dialog
dialogExists	there is a dialog with the same name as the menu item
disabled	item is disabled
enabled	item is enabled
fileExists *filename*	specified file exists on disk
firstInGroup	item is first radio button in its group
firstWindow	this is the first window
has editField	the window has an edit text field
has growBox	the window has a grow box
has hScroll	the window has a horizontal scroll bar
has mainScroll	the window has a scroll bar
has namedResources	the resource type has any named resources
has vScroll	the window has a vertical scroll bar
haveField	record or struct has at least one field
hierarchical	the menu is hierarchical
lastInGroup	item is last radio button in its group
main	the menu is in the main menu bar
main|hierarchical	menu is main or hierarchical
mainScroll	item is a scrollBar along window's edge

Variable	Boolean value
modal	dialog is modal
modalOneShot	dialog is palette dialog
modeless	dialog is modeless
multiWindow	there are multiple window kinds
needsFilter	dialog needs a filter proc
OK	item is item 1 in a modal dialog
variable	static text item has no text
window	item is in window (not dialog)

Boolean variables are used only in conditional code generation directives, to determine the nature or state of a particular user interface element. When a boolean variable is used in an AppMaker template, it will either have a TRUE or FALSE value at the time it is encountered and interpreted by AppMaker.

Boolean Expressions

AppMaker's Template Language contains a facility for specifying boolean expressions. The syntax for a boolean expression is shown below:

not *booleanExpression*
| *booleanVariable*
| *stringVariable = string expression*
| *numberVariable > number*

What this means is that a boolean expression can consist of the word "not" followed by a boolean expression, simply a boolean variable's name, an expression that tests the equality of a string variable against a string expresion, or a number variable whose value is compared to be greater than a constant.

String Expressions

String expressions are constructed by concatenating two or more string variables, string constants, or a combination of the two. The concatenation operator is the plus (+) character. Thus, a filename can be built up by concatenating the name of a window, menu, or other interface item with an appropriate file extension, as shown in the example below:

```
%genfile dialog U+dlogname+.p%
```

In this case, the filename will consist of the letter U, followed by the name of the current dialog, with the suffix .p, indicating a Pascal file.

Template Language Directives

The Template Language contains several different directives. As previously indicated, each directive is enclosed by percent sign (%) delimiters. The individual words of the directives described below are not case-sensitive, but are blank-sensitive. Make sure that each word in the directives you write is separated from the next by one and only one space character.

File Generation Directive

A complete file can be generated by using the genfile directive, whose syntax is shown below:

%genfile *templateName filename%*

The *templateName* parameter is the name of another template resource in the same set of templates. *Filename* parameter is a string expression (e.g. menuname+.c) which specifies the name of the generated source file.

For Each Directive

The `for each` directive repeats the interpretation of the following directives for each of the specified user interface items. The syntax of this statement is shown below:

```
%for each {menu | menuItem | window | dialog |
item | resource} gen caseLabel%
```

The curly braces and vertical bar characters are not typed, but are metacharacters that indicate that one (and only one) of the choices must be selected. The name of the template to interpret is implied by the chosen item following the words `for each`. The actual template name is the name of the choice, with the word Each prepended. The "case" to execute in that template is specified following the word "gen". For example, a directive of this type might read as follows:

```
%for each MenuItem gen doItem%
```

This directive specifies that for each menu item in the current menu, it should execute a case called doItem in the template called EachMenuItem.

Code Generation Cases

In the description of the `for each` directive, mention was made of the *case* to be executed. Many of the code generation templates contain `case` directives that are only interpreted when referenced by the `for each` directive.

A `case` directive has the following syntax:

```
%case caseLabel1%
      %<code to generate>%
%case caseLabel2%
      %<code to generate>%
%else%
      %<code to generate>%
```

The *caseLabel* parameter is the name of a case that will be referred to in a `for each` directive. All of the statements and directives following the `case` directive, up to the next `case` directive, (or the end of the template) are interpreted as being part of that case. The *<code to generate>* statement represents zero or more lines of code to be generated for that case. The `case` directive also has an optional `else` clause that permits code to be generated if a *caseLabel* other than those defined in the template is used. For example, using the example in the `for each` directive description, the code in the `EachMenuItem` template for the `doItem` case might read:

```
%case doItem%
  %if commandNr > 999%
    %if dialogExists%
      %if modeless%
        {$ ASelCommand}
        {----------}
        Procedure
                T%appname%App.Do%dlogname%;
        Begin
          f%dlogname%.Select;
          f%dlogname%.Show (true, true);
        End; {Do%MenuItemName%}
      %endif%
    %endif%
  %endif%
%else%
  {Invalid case %caseName%}
```

In this example, every line between the `%case doItem%`, and the `%else%` clause would be generated. If some other *caseLabel* were given, the template would generate a comment that specified the invalid *caseLabel*.

Code Generation for Resources

AppMaker's Template Language also has the ability to generate code that is specific to each resource type included in the user interface definition. A variation of the `for each` directive is used for this purpose. The syntax for this directive is shown below:

```
%for each resourceType gen templateName%
```

In this case a named template is provided, and the resources for which code is to be generated are indicated by case directives, whose case-names are identical to Macintosh resource type names. For example, a `for each resourceType` statement that names the template `DefineResources` might refer to cases in that template that contain the following code:

```
%case MENU%
   const {%RsrcType% Ids and commandNrs}
      %for each menu gen defineMenu%
%else%
   %if has namedResource%
      const {%RsrcType% IDs}
      %for each resource gen defineResource%
```

If Directives

The `if` directive provides the means to test the result of a boolean expression and generate code depending on whether it is TRUE or FALSE. As indicated in the section describing boolean expressions, the `not` keyword can be used to negate the result of the expression that follows it.

The syntax for the `if` directive is shown below:

```
%if booleanExpression%
   <code to generate>
[ %elsif booleanExpression%
   <code to generate> ]
[ %else%
   <code to generate> ]
%endif%
```

The <code to generate> statement following each clause of the if directive represents zero or more lines of code that are to be generated if the expression is true. The square brackets ([ ]) indicate that these clauses are optional and may be omitted if desired. A typical `if` directive might be coded as follows:

```
%if classname = TSScrollBar%
%elsif classname = TScrollBar%
%else%
   if Member (TObject (nil), %classname%) then
      ;
%endif%
```

This example is evaluated as follows: if the variable called classname is not equal to the string "TSScrollBar" and is also not equal to the string "TScrollBar", then the code beginning with "if Member (..." is generated.

Other Directives

There are also several Boolean directives which are assertions that set the corresponding boolean variable name to a TRUE value. The Boolean directives are: needsFilter, haveField, and multiWindow. An example of a Boolean directive that uses the haveField assertion is shown below:

```
%case dialog field%
   short%   %%fieldname%;
   %haveField%
```

In this example, when the "dialog field" case is executed, the field's name is defined, then the haveField boolean variable is set to TRUE.

Analyzing Some Real Templates

It would take a manual many times the size of this to contain an analysis of all the code generation templates that provide AppMaker with its flexibility. However, in this section we will analyze a set of templates to give you a general idea of how the Template Language is used to generate some real code.

For each language or variant you choose to generate code, AppMaker selects a set of template resources. In our case, we've selected THINK Pascal and the MacApp class library as our language, so the resource type for our templates is TmPM.

Regardless of the language (and therefore the resource type), the first template that is always executed is named GenWhat. This template, for our language, is shown below:

```
%if lang = MPW%
   %genfile MakeFile appfilename+.MAMake%
%elsif lang = Think%
   %genfile ---BuildOrder%
%endif%
   %genfile Main M+appname+.p%
   %genfile App U+appname+.p%
%if lang = MPW%
   %genfile App.inc U+appname+.inc.p%
%endif%
   %genfile Doc U+appname+Doc.p%
%if lang = MPW%
   %genfile Doc.inc U+appname+Doc.inc.p%
%endif%

%for each dialog gen sourcefile%
%for each window gen sourcefile%
   %genfile ResourceDefs ResourceDefs.p%
%if lang = MPW%
   %genfile RezFile appFileName+.r%
%endif%
```

As you can see, this template starts off by testing the boolean variable `lang` to determine if it is generating code for the MPW compiler environment. Because this is not the case, AppMaker skips the code inside that condition and test if the language is Think. This condition is true, so AppMaker interprets the code inside that condition.

The next line of code is an AppMaker directive that directs it to generate a new file, using the template called ---BuildOrder.

AppMaker hasn't forgotten that it is currently interpreting the GenWhat template, it merely calls the ---BuildOrder template like a subroutine to continue generating code. Note that we haven't yet generated a single line of code, but we will shortly.

The ---BuildOrder template is used only for THINK Pascal. It specifies the order in which the generated code modules should be added to THINK's project file. The content of this file is merely a set of comments that specify the names of the source files, in the order of their dependency to one another. Following is the contents of the template whose name is ---BuildOrder:

```
{ Start with the MacApp seed project shipped
with THINK Pascal.}
{ Add all generated files, as well as
UAMLibaryM.p, and then drag}
{ the AppMaker-generated files into the
following build order: }

{ THINK and MacApp files }

{  ---BuildOrder%    %}
{  UAMLibraryM.p%    %}
{  ResourceDefs.p%   %}
%for each dialog gen buildOrder%
%for each window gen buildOrder%
{  U%appname%Doc.p% %}
{  U%appname%.p%     %}
{  M%appname%.p%     %}

{  Be sure to designate the AppMaker document
as this project's }
{  resource file.  The procedure is as
follows: }
{      (1)Choose "Run Options…" from the Run
menu. }
```

```
{      (2) Click on the "Use Resource File"
check box. }
{      (3) Choose the AppMaker resource file
from the scrolling list.}
{      (4) Click the "Use" button. }
{      (5) Click the "OK" button. }

Unit BuildOrder;
Interface
Implementation
End.
```

Prior to beginning interpretation of the ---BuildOrder template, AppMaker creates and opens a new file with the name, ---BuildOrder.

The first series of lines in the generated ---BuildOrder file are Pascal comments that direct the user how to interpret the contents of this file. In particular, they instruct the user how to add the named files to the project and how to specify the name of the resource file to be copied into the application.

Notice also that directives can also be embedded inside comments. AppMaker doesn't know or care that text enclosed inside curly braces represents comments to the Pascal compiler. An example of this is the line:

```
{  ---BuildOrder%   %}
```

which merely inserts a number of Tab characters between "BuildORder" and "}". After generating the comments mentioning the ---BuildOrder file itself and the UAMLibraryM.p file, which contains library support routines for AppMaker applications, the next AppMaker directive encountered is:

```
%for each dialog gen buildOrder%
```

What this directive specifies is that for each dialog defined for the application, AppMaker is directed to interpret the template called EachDialog and execute the case named buildOrder (remember that in the `for each` directive, the word Each is prepended to the type of resource item, hence Each + Dialog creates the template name EachDialog). Following is the content of the buildOrder case from the EachDialog template:

```
%case buildOrder%
   {  U%dlogname%.p%  %}
```

Notice that the buildOrder case merely generates another comment, the contents of which is the letter U, followed by the name of the dialog (%dlogname%), the suffix .p, and Tab characters to make the right curly brace line up with the others in the file.

The buildOrder case is interpreted for each dialog that was defined in the user interface, so there could be several such comments generated in the file. The next directive in the ---BuildOrder file is the following:

```
%for each window gen buildOrder%
```

This instructs AppMaker to interpret the buildOrder case of the EachWindow template, which contains the following directives:

```
%case buildOrder%
   %if rsrcID > 999%
      {U%windname%.p%  %}
   %endif%
```

Notice that this case checks if the resource ID of the window is greater than the constant 999, by testing the Numeric variable rsrcID. (All user-specified windows should have a resource number [rsrcID] greater than the reserved range 1-999.)

If the test succeeds, then a comment containing the letter U, followed by the window's name (%windname%), the suffix .p. some explicit Tab characters, and a right curly brace are generated.

Following the generation of comments for each window, AppMaker will generate comments naming the file containing the document class, the application class, and the main program, by substituting the name of the application for the string variable %appname%. In the case of the AMReminder application discussed earlier in this manual, these filenames would be generated as AMReminderDoc.p, UAMReminder.p, and MAMReminder.p.

Interpretation of the ---BuildOrder template concludes by generating some additional comments and the following Pascal statements:

```
Unit BuildOrder;
Interface
Implementation
End.
```

These statements permit the ---BuildOrder file to be added to the project, the same as any other Pascal source file, and not cause the THINK Pascal compiler to generate error messages because it isn't complete.

Because interpretation of the ---BuildOrder template is now finished, AppMaker will return to interpreting the GenWhat template, exactly where it left off.

The very next directive in the GenWhat template instructs AppMaker to generate using the template named Main into a file named the letter M, followed by the name of the application (%appname%), with the suffix .p. In the case of our AMReminder application, this file would be named MAMReminder.p.

It is clear that the majority of the templates have not yet been described; however, interpretation of these follows the same rules as stated by the directive syntax and the examples just given.

It is very instructive to examine the templates, even if you're just interested in how AppMaker generates its code.

If you intend to customize the templates to suit your unique requirements, studying the existing templates will give you a very good idea of how to write directives that cause your changes to be generated.

If you want to make sweeping changes to the generated code (perhaps to support a different class library, or a completely different language), studying the existing templates will provide a good foundation for what you need to do to automatically generate the code you require.

Index

B • O • W • E • R • S
Development

P.O. Box 9, Lincoln Center, MA 01773
(508) 369-8175 • FAX 369-8224